

John Sweeney
Rochester Institute of
Technology
Rochester, New York
John.Sweeney@RIT.edu

Copyright 2013 by
John Sweeney
No rights reserved
Public Domain

Introductory Java 7 Computer Programming for Mathematicians

by

John Sweeney
Rochester Institute of Technology

Chapter One: First Java Program

Topic: Goal

I hope students of mathematics will find this book as a helpful start to their use of the java computer programming language.

Developing computer programming skills requires extensive experience using a computer programming language to solve problems. Being aimed at mathematicians this book has problems that exclusively focus on mathematics. Assuming a reader has completed introductory courses in algebra, trigonometry, and geometry this book contains references to mathematics easily comprehended by novice mathematicians. The set of problems do not focus on any area of mathematics, nor do they cover all of mathematics. The problems chosen for this book on java do not form an introduction to mathematics, nor any area of mathematics. Instead the chosen mathematical topics

and problems are intended to help the reader maintain interest in completing the exercises and complete this introduction to the java programming language. The problems chosen for this book emphasize the development of computer programming skills, specifically the java computer programming language.

Since 1995 the java computer language has proven itself as a good computer language for students learning computer programming techniques built on object-oriented concepts using a single processor. Amongst students java is notorious for requiring all computer programming variables to be defined before being used. This contrasts with many popular introductory computer programming languages such as Visual Basic in which the computer allows variables to hold any type of data and not even requiring their definition before you begin using a variable. The extensive structure and rules of java permit the computer compiler to detect your programming errors and provide a more appropriate error message. The compiler does not understand the purpose of your computer program, nor the student's approach to solving a problem. The compiler can only enforce the rules of the programming language, so all error messages can be cryptic.

Developing a computer program requires an extensive amount of time devoted to understanding the problem. Next the programmer should develop appropriate data sets to test the proposed computer program. Only then should coding begin. Testing and fixing errors in your computer program so much time

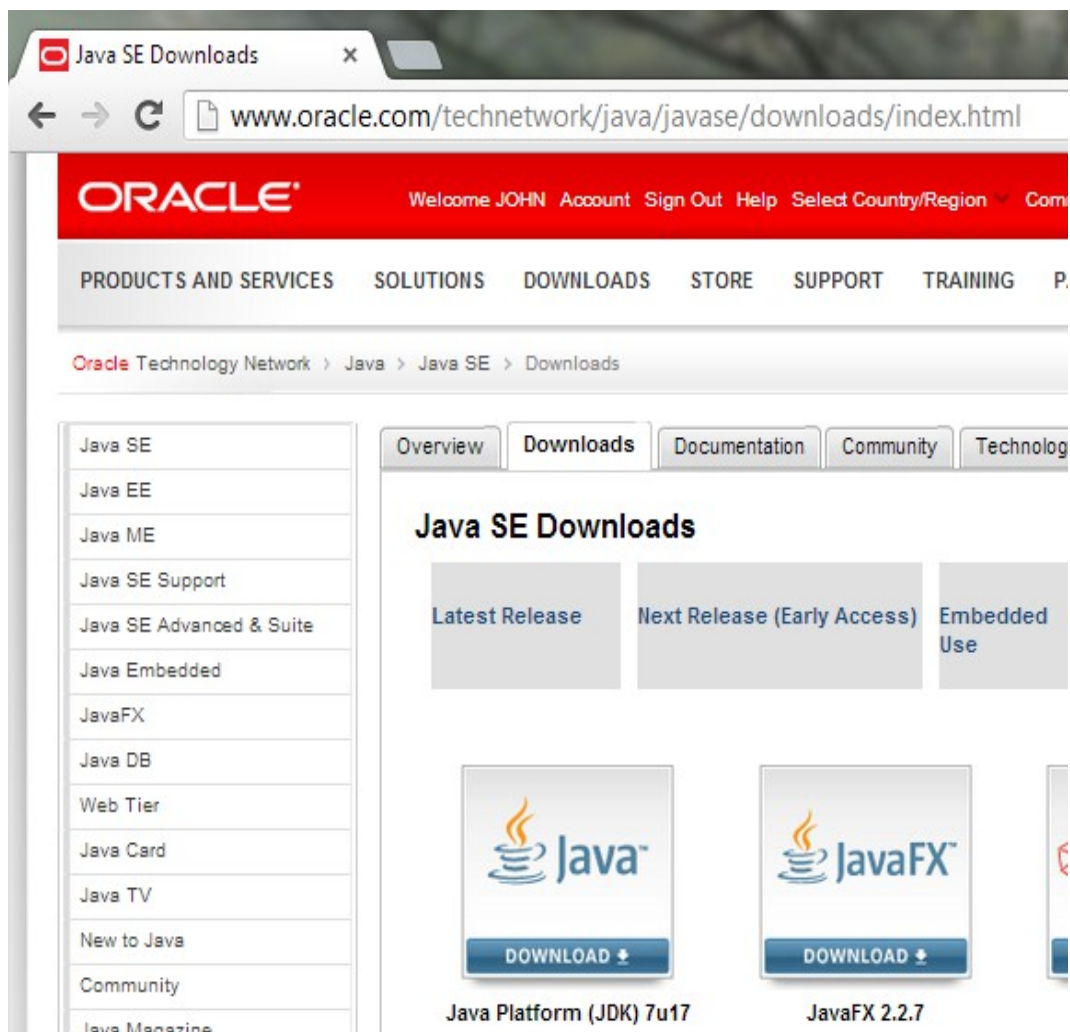
that it becomes the major portion of the entire development effort. Although java programs do require more typing the effort devoted to typing is a small portion of the entire project. Studying java as their first computer language has proven to assist students in their learning of additional computer programming languages.

Java has proven to be a difficult language to teach as a first computer programming language. Most books beg the reader to show patience since parts of the their first java programs will not be discussed or explained until a delay of several chapters. Every effort has been made to explain, although sometimes only in a cursory fashion, java constructs as soon as they appear in example programs. A simple java program involves a large number of concepts, so even a simple introductory java program can overwhelm novice java programmers.

This book emphasizes object-oriented programming concepts. Java provides good support for your study of those concepts. Example programs support the writing of object-oriented solutions which are easily comprehended by other computer programmers. Most computer development involves large teams of programmers and computer solutions to mathematical problems should be reviewed by other mathematicians so a clear programming style has importance to your future work.

Studying a computer language, especially java, requires a commitment of a large amount of your time

and energy. Carefully compare the effort to learn java with its benefits to your future work. Learning java simply to earn an academic grade will ensure that you will quickly forget everything about java. Completion of each chapter's exercises will aid in the development of introductory object-oriented programming skills using a single processor computer. Employing java as a tool to solve problems is a higher-level of learning. Such a large effort on your part may lead to long term retention of the core concepts of object-oriented programming using java.



The screenshot shows a web browser window with the title "Java SE Downloads". The address bar displays the URL "www.oracle.com/technetwork/java/javase/downloads/index.html". The Oracle logo is prominently displayed at the top, followed by a navigation bar with links for "PRODUCTS AND SERVICES", "SOLUTIONS", "DOWNLOADS", "STORE", "SUPPORT", "TRAINING", and "P...". Below this, a breadcrumb trail reads "Oracle Technology Network > Java > Java SE > Downloads".

The main content area is titled "Java SE Downloads" and features a sidebar on the left with a list of links: "Java SE", "Java EE", "Java ME", "Java SE Support", "Java SE Advanced & Suite", "Java Embedded", "JavaFX", "Java DB", "Web Tier", "Java Card", "Java TV", "New to Java", "Community", and "Java Magazine".

The main content area has tabs for "Overview", "Downloads", "Documentation", "Community", and "Technology". Under the "Downloads" tab, there are three sections: "Latest Release", "Next Release (Early Access)", and "Embedded Use". Below these, there are two large download buttons: "DOWNLOAD" for "Java Platform (JDK) 7u17" and "DOWNLOAD" for "JavaFX 2.2.7".

Topic: Preparation

Besides this book you will need the java development kit and a text editor or integrated development environment(IDE). You can download the java development kit from Oracle's Web site at <http://www.java.com>. A quick method to locate the appropriate file is to use a search engine, such as Google, with the search string of "download JDK". JDK is an abbreviation of java development kit. SE is the standard edition of java. Java SE 7 was used to develop the examples in this book. Although the screen image shown below includes the internet address of that download, it may change in the future, so a Web search for "download JDK 7" is preferable. You should always



Topic: IDE

The examples in this book were written using the jGRASP IDE available for download at **<http://www.jgrasp.org>**. Many good quality IDEs are available for your use, you should feel free to use any of them. jGRASP was developed for novice programmers and is used to develop small programming projects.

You should enter the java program shown as example 1 into your chosen IDE, then compile and run your first java program. Consult your IDE's manual for directions to complete that exercise.

The exercises in this book will benefit you best

if you write your own computer programs to further your exploration of mathematics. Only with practice can you make progress and develop expertise in writing java programs.

Example 1: The Source code of your first java computer language program

```
/**
 * The FamousMathQuotes class
 * This computer program prints a famous quote of
Carl Friedrich Gauss.
 */
class FamousMathQuotes {

    public static void main(String[] args) {
        System.out.println("Carl Friedrich Gauss
described mathematics as the Queen of the Sciences.");
// Display the string.
    }

}
```

Exercise: Search the internet for other famous sayings by mathematicians. Print a variety of famous sayings by mathematicians.

You type the **source code** for your computer program into the text area provided by your IDE.

Topic: Comments

Java programs have three ways to include comments: block comments, inline comments and end of line comments. This program uses a block comment. The top four lines of the **FamousMathQuotes** program are a block comment describing the purpose of the entire program. **/**** marks the start of that comment and ***/** marks its end. Everything between those markers is a comment. Notice that the beginning marker contains two asterisks, not one. Customarily IDEs automatically begin each new line of such comments with an asterisk and a space, but they are not required. Java provides two other methods for adding comments to your program's source code.

Topic: Commenting Conventions

The java community of programmers has developed extensive guidelines for the contents of comments. Portions of those guidelines will be introduced in later chapters. For now the beginning comment, which is known as a **doc comment**, satisfies its goal of describing the purpose of this java program. Doc comments provide descriptions of the program or method's goal, algorithms employed, and data structures used. Doc comments should be separated by a blank line from any preceding code. The first line begins with **/****. Following lines begin with an optional asterisk, *****, to reinforce the visual perception of being a line of comment. The final line begins with ***/**. Thus **/**** begins the doc comment and ***/** ends that doc comment.

```
/**
 *   Program goal, algorithms, and
 *   data structures are described here.
 */
```

Oracle provides more information at

<http://www.oracle.com/technetwork/java/codeconventions-141999.html#385>

The java compiler ignores all comments, which are written solely to help us comprehend the computer program. A second style of marker begins with `/*` and ends with `*/`. Programmers often use this style of comment when multiple lines of description are required to properly explain a portion of the computer program. The third style of comment begins with `//` and continues to the end of the line. Notice `// Display the string.` near the end of the example 1 program. Such comments are a popular method to include a short description of the command on the same line. Because the compiler will ignore all comments, you could correctly assume comments are not required, thus the same program could simple be written as shown in example 1A.

Topic: example java program 1A

```
class FamousMathQuotes {

    public static void main(String[ ] args) {
        System.out.println("Carl Friedrich Gauss
described mathematics as the Queen of the Sciences.");
    }
}
```

```
    }  
  
}
```

Topic: Class

The computer program begins with "class." A class describes an object, an example of such might be a robot, which would have attributes and abilities. By the standard naming convention we begin a class name with a capital letter. The left curly bracket, or open brace, symbol { begins the body of that class, whose end is marked by the right curly bracket, or closing brace, } symbol. Of course those symbols forms teams. Forgetting to match brackets is an error and will block the creation of the associated java program and its execution. Most IDEs will automatically align those symbols so you can easily notice the beginning and end of a class's body. There are two different standards for curly braces. The first standard puts the left curly brace { at the end of the class line with the closing right curly brace on a line by itself. Then the closing curly brace, } aligns with the class command, but the opening left curly brace does not align.

```
class FamousMathQuotes {  
...  
}
```

Move your cursor over such a symbol and your IDE might highlight the matching symbol. Attributes are described early in the class. This class has no

attributes.

Within the class you will notice "**public static void main(String[] args)**" which describes the method named "**main.**" In a world of objects the modifier "**public**" means other objects can see and use your method named "**main.**" Classes are used to make objects.

The modifier **static** simply means the method stays with the class and is not included with each object. Recall you write methods to implement the object's abilities, so our objects of type **FamousMathQuotes** would have no abilities. Thus any object made from class **FamousMathQuotes** has neither attributes nor methods. They would be objects in name only.

The modifier **void** means your method does not produce, or return, anything. All the action occurs with the command: **System.out.println("Carl Friedrich Gauss described mathematics as the Queen of the Sciences.");** **System** is the name of a special class containing attributes and methods of use generally to all java programs. **Out** is the output print stream, which is an attribute of the class **System**. Your IDE automatically connects that output print stream with the console of your IDE. For **jGRASP** users look at the bottom of **jGRASP's** window where you will find it under the tab labeled "Run I/O". "I/O" is an abbreviation of "input / output" meaning your java program prints to that area and there you can type replies to your program's questions. All java methods are followed by (and). The string "Carl Friedrich Gauss described mathematics as the Queen of the Sciences." must begin and end with double quotation marks.

Topic: Example java program 2

```
/**
 * The FamousMathQuotes class
 * simply prints a famous quote of Carl Friedrich
Gauss to standard output.
 */
class FamousMathQuotesV2 {

    /**
     * Method go prints
     * a famous quote of Carl Friedrich Gauss to
standard output.
     */
    private void go()
    {
        System.out.println("Carl Friedrich Gauss
described mathematics as the Queen of the Sciences.");
// Display the string.
    }

    /**
     * The main method is automatically executed.
     */
    public static void main(String[] args) {
        FamousMathQuotesV2 entity = new
FamousMathQuotesV2();
        entity.go();
    }
}
```

Study the second java program example. The class has been renamed **FamousMathQuotesV2** where **V2** means the second version. Each object has a method named **"go"** which prints Gauss' famous saying. That method is **private**, meaning only that object can use that method. The **FamousMathQuotes** class had no attributes nor methods. Our **FamousMathQuotesV2** entity now has an ability to print a famous saying.

Notice how the main method has changed. The print statement has been moved to the go method. Now you notice a command to create an object of type **FamousMathQuotesV2()** by using the **"new"** command followed by the name of the class. The new variable named **"entity"** is of type **FamousMathQuotesV2** so our definition is the type then the variable's name thus **"FamousMathQuotesV2 entity."** The equals command stores the address of our new object in variable **"entity."** The next line **"entity.go();"** calls the **go** method of the object. Then that **go** method will print our famous mathematician's saying.

Each method also has its own block comment. The doc comment begins with **/**** and ends with ***/**. Each method and class should begin with a doc comment. A doc comment begins with a brief description of the method or class. Then a list of **@param** statements describe each parameter. The **@return** statement describes the value to be returned by a method.

Obviously our example 2 program produces the same result as example 1. Example 2 demonstrates the role of an object. You can use either approach with your

own java programs.

Chapter Two: java primitives

Topic: Using java primitives to represent numbers

The followers of Pythagoras knew the first few perfect numbers which equal the sum of their proper positive divisors. For example 6 is the sum of 1, 2, and 3 which are all its proper positive divisors. Although 6 divides 6 it is not considered to be a proper divisor. For more information concerning perfect numbers you can consult Wikipedia at http://en.wikipedia.org/wiki/Perfect_number.

Topic: Example java program 1

```
/**
 * Print the sum of 6's proper positive divisors.
 */
public class SumPositiveDivisors {

    /**
     * This method prints the sum of 6's proper
     positive divisors.
     */
    private void go()
    {
        int divisor1 = 1;
        int divisor2 = 2;
        int divisor3 = 3;
        int sum = divisor1 + divisor2 + divisor3;
        System.out.println(sum); // Display the
answer.
    }
}
```

```

    /**
    * The main method is automatically executed.
    * Create an object of class SumTwoIntegers
    * then call its method named go.
    */
    public static void main(String[] args) {
        SumPositiveDivisors entity = new
SumPositiveDivisors();
        entity.go();
    }

}

```

In addition to objects the java language includes primitives, such as `int` to represent integers and `double` to represent real numbers. All numbers in a computer system are represented by binary numbers. An `int` has a size of four bytes. In java the byte is standardized at eight bits. An `int` contains 32 bits thus providing for integers from -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). If your work involves larger integers you can use the `long` primitive providing double the width for a size of 8 bytes or 64 bits and integer values from -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). For more information consult Oracle's Web site on java primitives at <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.

You can easily use `int` values in a command such as `sum = divisor1 + divisor2 + divisor3`, but first

java demands that you define each variable. A definition for `divisor1` includes `int`, the variable's type, and `divisor1`, the variable's name. You can also set that variable to an initial value by using the `=` operator. Thus `int divisor1 = 1;` defines a new `int` value with a name of `divisor1` and sets it to an initial value of 1.

A definition consists of the following: **type name = initial value;** We have seen this pattern in chapter one in the command: `FamousMathQuotesV2 entity = new FamousMathQuotestv2();` This program has examples such as `int divisor1 = 1;` and `int divisor2 = 2;`

Because the use of dynamic objects is considered to be a better programming paradigm our future examples will use that approach. Static methods and classes will be limited to supporting roles.

Topic: Computer program error

The java compiler might produce the error message "cannot find symbol" which confuses most novice programmers.

```
SumPositiveDivisorsV2.java:13: error: cannot find symbol
    int sum = divisor1 + divisor2 + divisor3;
              ^
symbol:   variable divisor1
location: class SumPositiveDivisorsV2
```

The line number 13 helps programmers find the source of the compiler error. The carat symbol `^` underneath `divisor1` also helps identify the source of

the compiler's complaint. Most development environments allow the programmer to display line numbers. Also the erroneous line will be highlighted.

```

1  /**
2  * Print the sum of 6's proper positive divisors.
3  */
4  public class SumPositiveDivisorsV2 {
5
6      /**
7      * This method prints the sum of 6's proper positive divisors.
8      */
9      private void go()
10     {
11         int divisor2 = 2;
12         int divisor3 = 3;
13         int sum = divisor1 + divisor2 + divisor3;
14         System.out.println(sum); // Display the answer.
15     }
16
17     /**
18     * The main method is automatically executed.
19     * Create an object of class SumTwoIntegers
20     * then call its method named go.
21     */
22     public static void main(String[] args) {
23         SumPositiveDivisorsV2 entity = new SumPositiveDivisorsV2();
24         entity.go();
25     }
26
27 }
28
29

```

Compilers begin by reading each line from the top of the class. The blanks neatly separate the line into symbols. Either the compiler recognizes the symbol as part of the java language or the compiler checks the list of previously defined variables. When the symbol cannot be found the compiler displays the message error: cannot find symbol. Programmers should check the spelling and check that all variables are defined before being used.

Topic: Another computer programming error

Here is another example of a computer programming error. The compiler cannot find the variables divisor1, divisor2, and divisor2.

```
▶ SumPositiveDivisorsV3.java:24: error: cannot find symbol
    int sum = divisor1 + divisor2 + divisor3;
                        ^
    symbol:    variable divisor1
    location: class SumPositiveDivisorsV3
▶ SumPositiveDivisorsV3.java:24: error: cannot find symbol
    int sum = divisor1 + divisor2 + divisor3;
                        ^
    symbol:    variable divisor2
    location: class SumPositiveDivisorsV3
▶ SumPositiveDivisorsV3.java:24: error: cannot find symbol
    int sum = divisor1 + divisor2 + divisor3;
                                    ^
    symbol:    variable divisor3
    location: class SumPositiveDivisorsV3
3 errors
```

The computer program is shown below.

```

1 /**
2  * Print the sum of 6's proper positive divisors.
3  */
4 public class SumPositiveDivisorsV3 {
5
6     /**
7      * The main method is automatically executed.
8      * Create an object of class SumTwoIntegers
9      * then call its method named go.
10     */
11     public static void main(String[] args) {
12         int divisor1 = 1;
13         int divisor2 = 2;
14         int divisor3 = 3;
15         SumPositiveDivisorsV3 entity = new SumPositiveDivisorsV3();
16         entity.go();
17     }
18
19     /**
20      * This method prints the sum of 6's proper positive divisors.
21      */
22     private void go()
23     {
24         int sum = divisor1 + divisor2 + divisor3;
25         System.out.println(sum); // Display the answer.
26     }
27
28
29 }
30
31

```

Those variables `divisor1`, `divisor2`, and `divisor3` are defined above the `go` method, but those variables are in the `main` method. During compilation the compiler keeps a table of all variables and their scope which means where the variable is known. Because `divisor1` is defined in method `main` its scope is method `main`. Only commands within `main` can use that variable. Thus the `go` method cannot use those variables.

Because variables `divisor1`, `divisor2`, and `divisor3` are used only in the `go` method those variables should be defined at the top of the `go` method.

Topic: Another programming error

This error message describes another way to mistakenly type your java computer program.

```

▶ SumPositiveDivisorsV4.java:25: error: cannot find symbol
    Entity.go();
      ^
  symbol:   variable Entity
  location: class SumPositiveDivisorsV4
1 error

```

Java identifiers are case sensitive. Thus Entity is not same identifier as entity. Notice that error on line 25.

```

1 /**
2  * Print the sum of 6's proper positive divisors.
3  */
4 public class SumPositiveDivisorsV4 {
5
6     /**
7      * This method prints the sum of 6's proper positive divisors.
8      */
9     private void go()
10    {
11        int divisor1 = 1;
12        int divisor2 = 2;
13        int divisor3 = 3;
14        int sum = divisor1 + divisor2 + divisor3;
15        System.out.println(sum); // Display the answer.
16    }
17
18    /**
19     * The main method is automatically executed.
20     * Create an object of class SumTwoIntegers
21     * then call its method named go.
22     */
23    public static void main(String[] args) {
24        SumPositiveDivisorsV4 entity = new SumPositiveDivisorsV4();
25        Entity.go();
26    }
27
28 }
29

```

Topic: java coding conventions

The java coding convention described at <http://www.oracle.com/technetwork/java/codeconv-138413.html> encourages programmers to begin the names of classes with a single capital letter. Variables should begin with lowercase letters. Thus **entity** should begin with a lowercase letter, not a capital letter.

Notice all three of the prior examples of program errors produced the same error message **cannot find symbol**. Only by understanding how the compiler works and knowing all the rules of the java programming language can you understand the cause of those errors.

Topic: Execution error

Find the error in the following java program's source code. The program sums 1, 2, and 3 and prints an answer of 5, not 6. No compiler error message is displayed and the program executes seemingly in a normal manner.

```
1 /**
2  * Print the sum of 6's proper positive divisors.
3  */
4 public class SumPositiveDivisorsV5 {
5
6     /**
7     * This method prints the sum of 6's proper
positive divisors.
8     */
9     private void go()
```

```

10    {
11        int divisor1 = 1;
12        int divisor2 = 2;
13        int divisor3 = 3;
14        int sum = divisor1 + divisor1 + divisor3;
15        System.out.println(sum); // Display the
answer.
16    }
17
18    /**
19     * The main method is automatically executed.
20     * Create an object of class SumTwoIntegers
21     * then call its method named go.
22     */
23    public static void main(String[] args) {
24        SumPositiveDivisorsV5 entity = new
SumPositiveDivisorsV5();
25        entity.go();
26    }
27
28 }

```

In this instance **divisor1** is shown where **divisor2** should be included in the list. Execution errors are notoriously difficult to spot and fix. Apparently people gloss over such slight errors and may even read the second **divisor1** as **divisor2**. We see what we intended to type, not the reality of a typo.

To detect such errors programmers produce by hand a list of the sample data, such as 1, 2, and 3, and the expected results of 6. Only by carefully comparing the computer program's result with our expectation can

we possibly detect an execution error.

Topic: approximations

Wikipedia lists historical approximations for pi at http://en.wikipedia.org/wiki/Approximations_of_pi where `%CF%80` is the code for pi. The following program calculates the values for approximations credited to Ahmes, the Babylonians, and Archimedes.

```
/**
 * Print three historical approximations of pi.
 */
public class ApproximatePi {

    /**
     * This method prints the sum of 6's proper positive
     divisors.
     */
    private void calc()
    {
        System.out.println("Math.pi: " + Math.PI);

        double ahmesPi = 256.0/81.0;
        System.out.println("Ahmes 256/81: \t" +
ahmesPi); // Display the answer.

        double babylonianPi = 25.0/8.0;
        System.out.println("Babylonian 25/8: " +
babylonianPi); // Display the answer.

        double archimedesPi = 22.0/7.0;
        System.out.println("Archimedes 22/7: " +
```

```

archimedesPi); // Display the answer.
    }

    /**
     * The main method is automatically executed.
     * Create an object of class ApproximatePi
     * then call its method named calc.
     */
    public static void main(String[] args) {
        ApproximatePi object = new ApproximatePi();
        object.calc();
    }
}

```

The program displays the following results.

```

Math.pi: 3.141592653589793
Ahmes 256/81: 3.1604938271604937
Babylonian 25/8: 3.125
Archimedes 22/7: 3.142857142857143

```

Notice how Amhes' approximation is 256/81 but our java computer program uses 256.0 / 81.0. The reason is that 256 is an **int** value while 256.0 is a **double** value. When java evaluates 256 / 56 both values 256 and 56 are **int** values, so the resulting answer will be an **int** value of 3. Likewise the Babylonian approximation of 25/8 would result in 3 and the Archimedes approximation of 22/7 would produce an **int** value of 3. Java would not produce any compiler errors nor warnings. The results will simply be 3.

Both 256.0 and 81.0 are **double** values so the result would be a **double** value. You might think that defining the result variable as a **double** would change the result, but it would not. The result would be a **double** value of 3.0 not an **int** value of 3. Java evaluates the expression to the right of the assignment operator (=) and assigns that result to the variable on the left of the assignment operator (=). The result of 256 / 81 is evaluated first as an **int** value of 3. Only later is that **int** value of 3 assigned to the **double** value of **ahmesPi** as 3.0.

The following erroneous version of our pi approximation program calculates 256/81.

```
/**
 * Print three historical approximations of pi.
 */
public class ApproximatePi {

    /**
     * This method prints the sum of 6's proper positive
     divisors.
     */
    private void calc()
    {
        System.out.println("Math.pi: " + Math.PI);

        double ahmesPi = 256/81;
        System.out.println("Ahmes 256/81: \t" +
ahmesPi); // Display the answer.
```

```

    double babylonianPi = 25.0/8.0;
    System.out.println("Babylonian 25/8: " +
babylonianPi); // Display the answer.

    double archimedesPi = 22.0/7.0;
    System.out.println("Archimedes 22/7: " +
archimedesPi); // Display the answer.
}

/**
 * The main method is automatically executed.
 * Create an object of class ApproximatePi
 * then call its method named calc.
 */
public static void main(String[] args) {
    ApproximatePi object = new ApproximatePi();
    object.calc();
}
}

```

The result of running the erroneous pi approximation program is shown here. The Ahmes pi variable contains 3.0.

```

Math.pi: 3.141592653589793
Ahmes 256/81:    3.0
Babylonian 25/8: 3.125
Archimedes 22/7: 3.142857142857143

```

When java must evaluate an expression composed of mixed types such `256.0 / 81` which contains a double value and an int value, the `int` value will first be

promoted to a **double**. Then the expression **256.0 / 81.0** will be evaluated producing a result of type **double**. Such promotions occur automatically without any compiler warnings.

When a value of type **double** must be assigned to a variable of type **int** the compiler will produce an error message because the fractional portion of that **double** value would be truncated thus information would be lost. If you wanted to know only the whole number portion of a double value use the cast command **(int)**. For example **Math.PI** is a value of type **double**, so a cast of type **int** is required for the following command. The compiler will notice the cast and avoid producing an error message.

```
int wholeIntPortion = (int)Math.PI;
```

When a number is too large to be stored for the variable's data type, that assignment will cause an overflow condition. Although the resulting value is certainly incorrect java does not produce any warning or errors.

```
/**  
* Print an example overflow condition for an int.  
*/  
public class OverflowError {  
  
    /**  
    * This method prints the sum of 6's proper positive  
divisors.  
    */
```

```

private void calc()
{
    System.out.println("maximum int: " +
2_147_483_647); // maximum int value 2,147,483,647

    System.out.println("maximum int + 1: : " +
( 2_147_483_647 + 1) ); // overflow.
}

/**
 * The main method is automatically executed.
 * Create an object of class OverflowError
 * then call its method named calc.
 */
public static void main(String[] args) {
    OverflowError object = new OverflowError();
    object.calc();
}
}

```

The execution of that program produces these erroneous results. The computer programmer must constantly consider the impact of potential overflow conditions and test results.

```

maximum int: 2147483647
maximum int + 1: : -2147483648

```

Many areas of recreational mathematics such as the search for prime numbers and square numbers have already produced results exceeding the size of an **int**

value. A lack of compiler errors and a seemingly successful execution of a computer program does not guarantee correct values. Computer programmers must be vigilant in checking the results of their computer programs.

Although ancient Egyptians, Mesopotamians, and Greeks used approximations to π , the entire field of approximate values is ignored in most books on mathematics. The comic XKCD gives us their own treatment of the topic at [**HTTP://XKCD.COM/1047/**](http://xkcd.com/1047/)

ARCHIVE
WHAT IF?
BLAG
STORE
ABOUT



A WEBCOMIC OF ROMANCE,
SARCASM, MATH, AND LANGUAGE.

LOTS OF EMAILS MENTION THE PHYSICIST FAVORITE, $1 \text{ YEAR} = \pi \times 10^7 \text{ SECONDS}$.
 75^4 IS A HAIR MORE ACCURATE, BUT IT'S HARD TO TOP 3,141,592'S ELEGANCE.

APPROXIMATIONS

|< < PREV RANDOM NEXT > >|

A TABLE OF
SLIGHTLY WRONG
EQUATIONS AND IDENTITIES
USEFUL FOR
APPROXIMATIONS
AND/OR
TROLLING TEACHERS
(FOUND USING A MIX OF TRIAL-AND-ERROR,
MATHEMATICA, AND ROBERT MUNARO'S *RIES* TOOL.)
ALL UNITS ARE SI MKS UNLESS OTHERWISE NOTED.

RELATION:		ACCURATE TO WITHIN:
ONE LIGHT-YEAR(m)	99^8	ONE PART IN 40
EARTH SURFACE(m ²)	69^8	ONE PART IN 130
OCEANS' VOLUME(m ³)	9^{19}	ONE PART IN 70
SECONDS IN A YEAR	75^4	ONE PART IN 400
SECONDS IN A YEAR (<i>RENT</i> METHOD)	$525,600 \times 60$	ONE PART IN 1400
AGE OF THE UNIVERSE (SECONDS)	15^{15}	ONE PART IN 70

Topic: Activity

Write your own java program to verify an approximation from Wikipedia's history of mathematics or XKCD.

Chapter Three: First Java Program

Topic: Classes

Java classes can represent mathematical objects such as a square. For more information concerning squares you can consult Wikipedia at <http://en.wikipedia.org/wiki/Square>. All squares are similar. They vary only in the length of the side. Although many descriptions of a square are possible, this chapter defines a class for a square solely by the length of its side.

When a class describes an object, that class has a name representing the object. A good name for such a class would be "Square." Thus the class's code begins as follows. Remember a class's name begins with a capital letter.

```
/**
 * This class represents a square in two dimensions.
 */
public class Square {

}
```

Each class begins with a list of attributes for the object it describes. Class Square needs only one attribute named **side** which we represent with a Java double primitive. A Java primitive is an item which is not an object. So far in this book, you have worked with the primitives **int** and **double**. Now the code for class **Square** appears as follows.

```

/**
 * This class represents a square in two dimensions.
 */
public class Square {

    double side;

}

```

Let **sq** be the name of a square of class **Square**. Your test program can easily access the contents of **sq**'s side by using the class name with the attribute in a command such as **System.out.println(sq.side);** Unfortunately a bad command such as **sq.side = -5;** could also easily insert a bad value. Programmers avoid such mistakes by making **side** a **private** attribute. Then only methods within class **Square** can access or change its contents. The accessor method normally begins with **get**. Mutator methods begin with **set**. Now class **Square**'s code contains an accessor and a mutator methods.

Topic: Example java program 1

```

/**
 * This class represents a square in two dimension.
 */
public class Square {

    private double side;
    private double area;

    /**

```

```
    * This default constructor makes objects of class
Square.

    */

    public Square() {
        side = 0.0;
        area = 0.0;
    }

    /**
    * This accessor method returns the length of the
square's side.
    *
    * @return      the length of the square's edge
    */
    public double getSide() {
        return side;
    }

    /**
    * This mutator method sets the length of the
square's side.
    *
    * @param  _side the new length for the square's edge
    */
    public void setSide(double _side) {
        side = _side;
        area = side * side;
    }

    /**
    * This accessor method returns the area of the
square.
    *
    */
```

```

    * @return      the area of the square
    */
    public double getArea() {
        return area;
    }
}

```

Suppose you had made a square of class Square with the name of `square`. `System.out.println("The square has an edge of length " + square.getSide());` would display the value of attribute `side`. `square.setSide(5)` would set the square to a size of 5 by 5. The list of arguments for a method is shown in the parentheses. The argument for `setSide` method has the new size of the square. A common name would be `side`, but that name would cause confusion because the class attribute is already named `side`. You might use a different name such as `newSizeValue` or simply begin with name with an underscore `_size`. Java recognizes `size` and `_size` as different names representing different values.

A different solution to avoid confusion uses the special identifier `this` for the current object. "`this.side`" means the object's attribute named `side`. "`side`" automatically refers to the closest identifier which is in the same method's list of arguments. The `setSide` method uses `this.side = side;` as shown below.

Topic: Example java program 2

```

//**

```

```

* This class represents a square in two dimensions.
*/
public class Square {

    private double side;

    /**
     * This accessor method returns the length of the
     square's side.
     *
     * @return double    the area of the square
     */
    public double getSide() {
        return side;
    }

    /**
     * This mutator method sets the length of the
     square's side.
     *
     * @param side the new length for the square's edge
     */
    public void setSide(double side) {
        this.side = side;
    }
}

```

Notice java allows two different variables to have the same name: **side**. The class **Square** has an attribute **side** known throughout the entire class, but not outside that class. Likewise the list of arguments of method **setSide** has a variable also named **side**, which is known throughout that method, but not outside that method. Confusion can occur because the range of

the class **Square** includes the method **setSide**. Within method **setSide** both the attribute **side** and the argument **side** exist. There they share the same name, **side**, but have different meanings. To avoid that confusion java automatically uses the nearest enclosing definition. Thus any reference to **side** within method **setSide** automatically means the argument, not the class attribute. Most programmers avoid this concern by using different names. For example the following class has been changed to use the parameter name of **_side**.

```
/**
 * This class represents a square in two dimension.
 */
public class Square {

    private double side;

    /**
     * This default constructor makes objects of class
    Square.
    */
    public Square() {
        side = 0.0;
    }

    /**
     * This accessor method returns the length of the
    square's side.
     *
     * @return      the length of the square's edge
     */
}
```

```

    public double getSide() {
        return side;
    }

    /**
     * This mutator method sets the length of the
    square's side.
     *
     * @param _side the new length for the square's edge
     */
    public void setSide(double _side) {
        side = _side;
        area = side * side;
    }
}

```

The technical word, *scope*, means the part of the class where the variable is known. `_side` is known only in method `setSide`. Attribute `side` is known throughout the entire class including method `setSide`.

Class `square` could also include an attribute for the area of the square. The following example program includes that new attribute named `area`.

```

    /**
     * This class represents a square in two dimension.
     */
    public class Square {

        private double side;
        private double area;
    }
}

```

```
/**
 * This default constructor makes objects of class
Square.
 */
public Square() {
    side = 0.0;
    area = 0.0;
}

/**
 * This accessor method returns the length of the
square's side.
 *
 * @return      the length of the square's edge
 */
public double getSide() {
    return side;
}

/**
 * This mutator method sets the length of the
square's side.
 *
 * @param  _side the new length for the square's edge
 */
public void setSide(double _side) {
    side = _side;
    area = side * side;
}

/**
 * This accessor method returns the area of the
```

```

square.
    *
    * @return      the area of the square
    */
    public double getArea() {
        return area;
    }
}

```

Now the **setSide** method performs a double duty of setting the length of the square's side and the square's area. Normally methods produce only a single result, so this is not the ideal design for this class. A better approach would rewrite the **getArea** method to access a pseudo-attribute. Instead of simply returning the value of `area`, the **getArea** method will calculate and return that result. The class `Square` now appears to contain another attribute named `area`. Such attributes are sometimes called pseudo-attributes. Many IDEs will automatically generate the javadoc for a class.

Class Square

java.lang.Object
Square

```
public class Square  
extends Object
```

This class represents a square in two dimension.

Field Summary

Fields

Modifier and Type	Field and Description
private double	side

Constructor Summary

Constructors

Constructor and Description
<code>Square()</code> This default constructor makes objects of class Square.

Method Summary**Methods**

Modifier and Type	Method and Description
double	<code>getArea()</code> This pseudo-accessor method returns the area of the square.
double	<code>getSide()</code> This accessor method returns the length of the square's side.
void	<code>setSide(double _side)</code> This mutator method sets the length of the square's side.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail**side**

`private double side`

Constructor Detail**Square**

`public Square()`

This default constructor makes objects of class Square.

When you create an object of class Square you use command **new**. An example command would be **Square obj = new Square();** The command automatically calls the class constructor. Constructors are special methods usually written immediately after the class attributes. Constructors must have the **public** attribute and not return any value. Do not even use a return type of **void**. Default constructors normally set initial values for all the class attributes. **Ints** are set to 0. **Doubles** to 0.0 and **Strings** to null or `""`. Thus class **Square** has the default constructor shown below.

```
/**  
 * This default constructor makes objects of class  
 Square.  
 */  
public Square() {  
    side = 0.0;  
}
```

Method Detail

getSide

```
public double getSide()
```

This accessor method returns the length of the square's side.

Returns:

the length of the square's edge

setSide

```
public void setSide(double _side)
```

This mutator method sets the length of the square's side.

Parameters:

_side - the new length for the square's edge

getArea

```
public double getArea()
```

This pseudo-accessor method returns the area of the square.

Returns:

the area of the square

`Square obj = new Square();` makes a square with name `obj` and its attribute `side` has the value `0.0`.

Classes also have a fully parametrized constructor to properly set the initial values for all attributes.

```
/**
 * This fully parametrized default constructor makes
 * objects of class Square.
 */
public SquareV2(double _side) {
    side = _side;
}
```

`Square obj2 = new Square(5.2);` makes a square with name `obj2` and its attribute `side` has the value `5.2`.
Class `Square` now has the code shown below.

Topic: Example java program

```
/**
 * This class represents a square in two dimensions.
 */
public class Square {

    private double side;

    /**
     * This default constructor makes objects of class
     * Square.
     */
    public Square() {
```

```

        side = 0.0;
    }

    /**
     * This fully parametrized default constructor makes
     objects of      class Square.
    */
    public Square(double _side) {
        side = _side;
    }

    /**
     * This accessor method returns the length of the
     square's side.
    */
    public double getSide() {
        return side;
    }

    /**
     * This mutator method sets the length of the
     square's side.
    */
    public void setSide(double _side) {
        side = _side;
    }
}

```

Beside the length of the side squares have a list of other properties. Such a list might include area, parameter, diagonal, location of its center, the surrounding circle, etc. You could write the Java code to calculate each of those other properties in your test class, or class **Square** could include methods to return the results of those computations. Then those

new methods act as pseudo-attributes. For area you might include an accessor, **get**, method as shown below.

```

    /**
     * This accessor method returns the area of the
     square.
     */
    public double getArea() {
        // return side * side; or side raised to the
    power of 2
        return Math.pow(side,2);
    }

```

The class **Math** is always available for your use. **Math** contains many useful methods such as **pow** that raises a **double** to a power. Searching the Java API for the **Math** class you would find the **pow** method.

static double	pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.
---------------	----------------------------------------------------------------------------------------------------------------

The method **pow** is static so our command begins with the name of the class **Math**. Squaring a number means to raise that number to the power 2, so our command is **Math.pow(side,2)**.

A square's perimeter is four times the length of a side, so the following method will perform that computation.

```

    /**
     * This accessor method returns the area of the
     square.
     */
    public double getPerimeter() {
        final int NUMBER_SIDES = 4;
        return side * NUMBER_SIDES;
    }

```

```
}
```

Notice the **final** attribute at the start of the declaration for **NUMBER_SIDES**. **Final** limits the program to exactly one change to the value of that variable. Because **NUMBER_SIDES** is set to 4, it cannot be changed again. Such constants have names in all capitals with the words separated by underscore characters. That declaration is included in method **getPerimeter** because **NUMBER_SIDES** is used only in that method. Because **NUMBER_SIDES** is inside method **getPerimeter** the other methods cannot use that constant. In technical terminology the scope of **NUMBER_SIDES** is the method **getPerimeter**.

Topic: Example java program

```
/**
 * This class represents a square in two dimensions.
 */
public class Square {

    private double side;

    /**
     * This default constructor makes objects of class
     Square.
     */
    public Square() {
        side = 0.0;
    }

    /**
     * This fully parametrized default constructor makes
     objects of class Square.
```

```
*/  
public Square(double _side) {  
    side = _side;  
}  
  
/**  
 * This accessor method returns the length of the  
square's side.  
 */  
public double getSide() {  
    return side;  
}  
  
/**  
 * This mutator method sets the length of the  
square's side.  
 */  
public void setSide(double _side) {  
    side = _side;  
}  
  
/**  
 * This accessor method returns the area of the  
square.  
 */  
public double getArea() {  
    // return side * side;  
    return Math.pow(side,2);  
}  
  
/**  
 * This accessor method returns the area of the  
square.  
 */  
public double getPerimeter() {  
    final int NUMBER_SIDES = 4;  
    return side * NUMBER_SIDES;  
}
```

```
}
```

```
}
```

Topic: Exercise

Add a pseudo-attribute for the square's hypotenuse by writing the `getDiagonal` method. Use the `hypot` method of the `Math` class to calculate that diagonal's length.

<code>static double</code>	<code>hypot(double x, double y)</code> Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
----------------------------	------------------------------------------------------------------------------------------------------------------

Next, we examine the test class `TestSquare` that will create an object of class `Square` and exercise its methods. As in previous programs that test class will create an object of its own class and execute its `go` method as shown below.

```
/**
 * This class tests class Square which represents a
 * square in two dimension.
 */
public class TestSquare {

    /**
     * This method creates an object of class Square
     * and displays information concerning that square.
     */
    private void go()
    {
        System.out.println("Create an object of class
Square.");
    }
}
```

```

/**
 * The main method is automatically executed.
 * Create an object of class TestSquare
 * then call its method named go.
 */
public static void main(String[] args) {
    TestSquare entity = new TestSquare();
    entity.go();
}
}

```

The go method will create an object of class Square and print information about the area and perimeter of that square. Next the program creates a square with an edge of length 5.2.

```

/**
 * This class tests class Square which represents a
 * square in two dimension.
 */
public class TestSquare {

    /**
     * This method creates an object of class Square
     * and displays information concerning that square.
     */
    private void go()
    {
        Square sq = new Square();
        System.out.println("Default square side: " +
sq.getSide() +
        " area: " + sq.getArea() +
        " perimeter: " + sq.getPerimeter()); //

```

Display the answer.

```
        Square sq2 = new Square();
        sq2.setSide(5.2);
        System.out.println("Second square side: " +
sq2.getSide() +
            " area: " + sq2.getArea() +
            " perimeter: " + sq2.getPerimeter()); // Display
the answer.
    }

    /**
     * The main method is automatically executed.
     * Create an object of class TestSquare
     * then call its method named go.
     */
    public static void main(String[] args) {
        TestSquare entity = new TestSquare();
        entity.go();
    }
}
```

In the previous program we did two steps to set up a square with edge length of 5.2.

```
Square sq2 = new Square();
sq2.setSide(5.2);
```

By providing a second constructor we can reduce that to one command: `Square sq2 = new Square(5.2);`
The new constructor accepts the size of the square as an argument.

```
/**
 * This fully parametrized constructor makes objects
of class Square.
```

```

*/
public SquareV2(double _side) {
    side = _side;
}

```

In addition to objects the java language includes primitives, such as **int** to represent integers and **double** to represent real numbers. All numbers in a computer system are represented by binary numbers. An **int** has a size of four bytes. In java the byte is standardized at eight bits, thus an **int** contains 32 bits providing for integers from -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). If your work involves larger integers you can use the **long** primitive providing double the width for a size of 8 bytes or 64 bits and integer values from -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). For more information consult Oracle's Web site on java primitives at <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.

You can easily use **int** values in a command such as **sum = divisor1 + divisor2 + divisor3**, but java demands that you define each variable. A definition for **divisor1** includes **int**, the variable's type, and **divisor1**, the variable's name. You can also set that variable to an initial value by using the **=** operator. Thus **int divisor1 = 1;** defines a new **int** value with a name of **divisor1** and sets it to an initial value of 1.

A definition consists of the following: **type name = initial value**; We have seen this pattern in chapter one in the command: **FamousMathQuotesV2 entity = new FamousMathQuotestV2()**; This program has examples such as **int divisor1 = 1**; and **int divisor2 = 2**;

Although Java automatically upgrades an **int** to a **double**, it will not automatically downgrade a **double** to an **int**. When you sum or multiply two **int** values the result is an **int** value. Likewise when all the variables are doubles the result will be a double. For a command that mixes an **int** value with a double value, Java will automatically upgrade the **int** value to a double before computing the result. For example **double x = 5 + 11.3**; the 5 will be upgraded to 5.0, a double value. Then that 5.0 will be summed with 11.3 to produce a result of 16.3 to be stored in variable x.

If you want only the integer portion of the result you can cast the answer to an **int** value by using the cast operator (**int**). For example **int rating = (int)(5 / 3.9)**; will upgrade the 5 to 5.0, then divide 5.0 by 3.9 producing an answer of 1.282051282051282 which the cast operator (**int**) will truncate to 1. Be aware that the (**int**) cast will truncate, not round, the result. If you want to round the result then use the round method of the Math class.

static [rint](#)(double a)

double	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
	<u>round</u> (double a)
static	
long	Returns the closest long to the argument, with ties rounding up.

Those methods, **rint** and **round**, do not return an **int** value, so a cast of (**int**) would still be required.

Topic: Inheritance

Use **extend** to expand upon an existing class. The new class will have its own method plus access to all the methods and attributes of its super class. For example class ColorSquare can expand upon class Square by including an attribute for color.

```
import java.awt.Color;

/**
 * This class represents a square in two dimension.
 */
public class ColorSquare extends Square {

    private Color color;

    /**
     * This default constructor makes objects of class
     Square.
     */
    public ColorSquare() {
        color = Color.RED;
    }
}
```

```
/**
 * This accessor method returns the length of the
square's side.
 */
public Color getColor() {
    return color;
}

/**
 * This mutator method sets the length of the
square's side.
 */
public void setColor(Color _color) {
    color = _color;
}

}
```

Color is a class within java, so the program's code begins with an import statement to allow java access to the Color class store in the awt directory. Awt is an abbreviation of abstract window toolkit and contains many utility classes. Often programmers use the asterisk operand to allow access to all classes within the awt directory. Then the import command is `import java.awt.*;` This broader directive will not make your java program bigger; java will import only the classes needed for your program.

Color contains int values from 0 to 255 for each of three primary colors: red, green, and blue according to the additive color theory established by

the computer industry and known as sRGB. For further information go to http://en.wikipedia.org/wiki/SRGB_color_space. The Color class has static attributes for common colors such as Color.RED. All objects are extensions of one object name Object, which has a toString method which returns a string containing the object's name and address in memory. The Color class defines its own toString method which displays the three values for red, green, and blue. For Color.RED toString displays java.awt.Color[r=255,g=0,b=0], so red has a maximum value of 255 and both blue and green have values of zero, so they are not displayed.

The test class includes commands to set the square's color and to display that color.

```
import java.awt.Color;

/**
 * This class represents a triangle in two dimension.
 */
public class TestColorSquare {

    /**
     * This method creates an object of class Square
     * and displays information concerning that square.
     */
    private void go()
    {
        ColorSquare sq = new ColorSquare();
        System.out.println("Default square side: " +
sq.getSide() +
```

```

" area: " + sq.getArea() +
" perimeter: " + sq.getPerimeter() +
" color: " + sq.getColor()); // Display the answer.
ColorSquare sq2 = new ColorSquare();
sq2.setSide(5.2);
sq2.setColor(Color.RED);
System.out.println("Second square side: " +
sq2.getSide() +
" area: " + sq2.getArea() +
" perimeter: " + sq2.getPerimeter() +
" color: " + sq2.getColor()); // Display the answer.
}

/**
 * The main method is automatically executed.
 * Create an object of class SumTwoIntegers
 * then call its method named go.
 */
public static void main(String[] args) {
    TestColorSquare entity = new TestColorSquare();
    entity.go();
}
}

```

Program execution displays the color as shown below.

```

Default square side: 0.0 area: 0.0 perimeter: 0.0
color: java.awt.Color[r=255,g=0,b=0]
Second square side: 5.2 area: 27.040000000000003
perimeter: 20.8 color: java.awt.Color[r=255,g=0,b=0]

```

Practice using other named colors. The javadocs for the Color class contains a list of the predefined colors such as **Color.BLUE** and **Color.WHITE**.

Topic: ArrayList

An **ArrayList** is a Java language object for storing other objects in a list. The javadocs page for ArrayList has the Internet address of <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>.

The name of the ArrayList object is squares, so a declaration is ArrayList squares; To declare and create that object named squares in one Java command is possible as shown here.

```
ArrayList squares = new ArrayList();
```

The newly created ArrayList has no content. The variable named squares is a pointer to an empty list. Our next commands create objects of class Square and add them to ArrayList squares.

```
Square sq1 = new Square(1.2);  
squares.add(sq1);  
Square sq2 = new Square(0.5);  
squares.add(sq2);  
Square sq3 = new Square(2.3);  
squares.add(sq3);
```

Topic: for each loop

Next our program defines a new variable of type double and name of sum to hold the answer as we sum the areas of the three squares.

```
double sum = 0.0;
```

The "for each" loop will process each of those three squares in ArrayList squares. The "for each" loop begins with the type and name of a variable to hold an element of the array. Next a colon separates that variable from the name of the array.

```
for (Square sq: squares)
{
    sum = sum + sq.getArea();
}
```

The general format is shown below.

```
for (type element: collection)
{
    statement
}
```

A collection is a java class for holding items. Programs use an extension of that collection class such as ArrayList which maintains a list of items. ArrayList is a type of Java collection, so the general format used "collection", not array.

The value of sum is replaced with the sum's current total plus the area of the next array element. That command can be abbreviated to **sum += sq.getArea();** In general all Java statements of the form **variable = variable + number;** can be abbreviated using the += operand. That abbreviation also works

with subtraction, multiplication, and division.

ArrayList objects can contain any objects. Because this program only stores objects of class Square the array declaration should be limited to holding only objects of class Square as shown below.

```
ArrayList<Square> squares = new  
ArrayList<Square>();
```

The Java program begins with an import statement. `import java.util.ArrayList;` So, the java compiler can obtain the code for an arrayList from that location.

The entire program is shown here.

```
import java.util.ArrayList;  
  
/**  
 * This class sums the area of several objects of class  
 * Square.  
 */  
public class SumSquares {  
  
    /**  
     * This method creates several objects of class  
     * Square  
     * and displays information concerning those  
     * squares.  
     */  
    private void go()  
    {  
        ArrayList<Square> squares = new  
ArrayList<Square>();
```

```

    Square sq1 = new Square(1.2);
    squares.add(sq1);
    Square sq2 = new Square(0.5);
    squares.add(sq2);
    Square sq3 = new Square(2.3);
    squares.add(sq3);
    double sum = 0.0;
    for (Square sq: squares)
    {
        sum = sum + sq.getArea();
    }
    System.out.println("Total area: " + sum);
}

/**
 * The main method is automatically executed.
 * Create an object of class SumSquares
 * then call its method named go.
 */
public static void main(String[] args) {
    SumSquares entity = new SumSquares();
    entity.go();
}
}

```

The program produces this result: **Total area:
6.979999999999999**

Because ColorSquare extends Square objects of class ColorSquare are also objects of class Square. Our test program would also work with sq3 of class ColorSquare.

The second version of our test program has sq3 as an object of class ColorSquare. Notice that this

version also includes an import statement for the Color class.

```
import java.awt.Color;
import java.util.ArrayList;

/**
 * This class sums the area of several objects of class
 * Square.
 */
public class SumSquaresV2 {

    /**
     * This method creates several objects of class
     * Square
     * and displays information concerning those squares.
     */
    private void go()
    {
        ArrayList<Square> squares = new
ArrayList<Square>();
        Square sq1 = new Square(1.2);
        squares.add(sq1);
        Square sq2 = new Square(0.5);
        squares.add(sq2);
        ColorSquare sq3 = new ColorSquare();
        sq3.setSide(5.2);
        sq3.setColor(Color.BLUE);
        squares.add(sq3);
        double sum = 0.0;
        for (Square sq: squares)
        {
            sum = sum + sq.getArea();
        }
    }
}
```

```

    }
    System.out.println("Total area: " + sum);
}

/**
 * The main method is automatically executed.
 * Create an object of class SumSquares
 * then call its method named go.
 */
public static void main(String[] args) {
    SumSquaresV2 entity = new SumSquaresV2();
    entity.go();
}

}

```

The second version of our test program produces this result: Total area: 28.730000000000004

Topic: IF statement

The next challenge is to find the size of the largest square in our collection named squares. The program begins with a declaration for the variable to hold the size of the largest square. That variable always begins with the smallest possible value, which for area is zero.

```
double largest = 0;
```

In the “for each” loop the if statement compares the current element's area with the value in variable largest. If it is bigger then the program stores the

new value in variable largest.

```
double largest = 0;
for (Square sq: squares)
{
    if (sq.getArea() > largest)
    {
        largest = sq.getArea();
    }
}
System.out.println("Area of largest square: " +
largest);
```

Our program prints the answer **Area of largest square: 5.289999999999999**

Next our program searches for the smallest area of the squares in our collection. The declaration for our variable to hold that answer is double smallest; which is set to Double.MAX_VALUE because the program is searching for the smallest value for area. Read the documentation for the class named Double. The static variable MAX_VALUE has the largest positive value for a double.

Field Summary	
Fields	
Modifier and Type	Field and Description
static int	<code>MAX_EXPONENT</code> Maximum exponent a finite double variable may have.
static double	<code>MAX_VALUE</code> A constant holding the largest positive finite value of type double, $(2-2^{-52}) \cdot 2^{1023}$.

```
double smallest = Double.MAX_VALUE;
for (SquareV2 sq: squares)
```

```

{
    if (sq.getArea() < smallest)
    {
        smallest = sq.getArea();
    }
}

System.out.println("Area of smallest square: " +
smallest);

```

The program prints **Area of smallest square: 0.25**

Topic: Exercise

Construct a class for rectangles with attributes of length and width. Use that class to implement the 12 orthogons of Wersin as described in Wikipedia's article on Dynamic Rectangles at http://en.wikipedia.org/wiki/Dynamic_rectangle. Show that those rectangles taken together can indeed have an area equal to a double square as described in that article.

"The 12 orthogons of Wersin

According to [Wolfgang von Wersin's](#) *The Book of Rectangles, Spatial Law and Gestures of The Orthogons Described* (1956), a set of 12 special *orthogons* (from the Gr. *ορθος*, *orthos*, "straight"[\[9\]](#) and *γωνια*, *gonia*, "angle"; "a right angled figure", which, as a consequence, is [rectangular](#) and [tetragonal](#)[\[10\]](#)) has been used historically by artists, architects and calligraphers to guide the placement and interaction of elements in a design.[\[3\]](#)[\[11\]](#) These orthogons are: [\[12\]](#)

- Square (1:1 or 1: $\sqrt{1}$)
- Diagon (1: $\sqrt{2}$)
- Hecton or sixton (1: $\sqrt{3}$)
- Doppelquadrat (1:2 or 1: $\sqrt{4}$)
- Hemiolion (2:3)
- Auron (the [golden rectangle](#), 1: φ)
- Hemidiagon (1: $\frac{1}{2}\sqrt{5}$)
- Penton (1: $\sqrt{\varphi}$)
- Trion (1: $\frac{2}{3}\sqrt{3}$)
- Quadriagon
- Biauron (1:2 φ)
- Bipenton

Wolfgang Von Wersin's book includes an extraordinary copy of text from the year 1558 ([Renaissance](#)), with diagrams of seven of the 12 orthogons and an invitation from the passage to pay careful attention as the "ancient" architects believed "nothing excels these proportions" as "a thing of the purest abstraction."[\[13\]](#)

All 12 orthogons, when formed together, create an entire unit: a square that is developed into a double square.[\[14\]](#)

Perhaps the most popular among the ortogons is the *auron* or [golden rectangle](#), which is produced by projecting the diagonal that goes from the middle point of a side of a square to one of the opposite vertexes, until it is aligned with the middle point.

Four of these orthogons are harmonic rectangles: the *diagon* or [root-2 rectangle](#) is produced by projecting the diagonal of a square; the *sixton*, *hecton* or [root-3 rectangle](#) is produced by projecting the diagonal of a diagon; the double square or [root-4 rectangle](#) is produced by projecting the diagonal of an hecton; the [root-5 rectangle](#) is produced by projecting the diagonal of a double square (or by projecting 180° both diagonals that go from the middle point of a side of a square to the

opposite vertexes).

Two of the most complicated of these figures are; the *penton*, with proportions $1:\sqrt{\phi}$ is related to the section of the [golden pyramid](#), the *bipenton*'s longer side is equal to the shorter multiplied by two thirds of the square root of three, longer side of the *biauron* is $\sqrt{5} - 1$ or 2τ times the shorter.

The *quadriagon* is related to the diagonal in the sense that its longer side is produced by projecting the diagonal of a quarter of a square. The *trion* has the height of an equilateral triangle and the width of the side. The *hemidiagon* ($1:\frac{1}{2}\sqrt{5}$) longer side is half the one of the root-5 rectangle and is produced by projecting the diagonal of half a square until it is perpendicular with the origin.

Besides the square and the double square, the only other static rectangle included in the list is the *hemiolion*, which is produced by projecting 90° or 180° half the side of a square." Source: Wikipedia Dynamic Rectangle.

Because the use of dynamic objects is considered to be a better programming paradigm our future examples will use that approach. Static methods and classes will be limited to supporting roles.

Chapter Four: java interface

Topic: Polygonal Numbers

Triangular numbers 0, 1, 3, and 6 can be represented by 0, 1, 3, and 6 dots arranged in triangles.

Wikipedia provides more detailed information at http://en.wikipedia.org/wiki/Triangular_numbers.

http://en.wikipedia.org/wiki/Figurate_number gives an equation for calculating the size of each triangular number of number n as $n(n+1)/2$.

Class **TriangleNumber** represents a triangular number. Each triangular number has a sequential ID as shown in the diagram above. Mathematicians commonly name each triangular number T followed by its ID. So the names are T_0 , T_1 , etc. Those ID numbers are positive integers beginning with zero. T_0 is the empty set, thus not shown in most drawings. The ID is an attribute of class **TriangleNumber**, so it is private, is set to zero by the default constructor, set to a positive integer by the fully parametrized constructor, and has an public accessor. Because a triangular number's ID never changes, attribute ID does not have an associated mutator method. This change from the norm of an accessor and a mutator for each attribute justifies the inclusion of a comment within the program's source code.

```
/**
```

```
 * Triangular numbers can be represented by a group of
```

```
dots
    * organized in an equilateral triangle.
    * The first few triangular numbers are of size 0, 1,
3,
    * 6, and 10. Those are the sizes of
    * triangles with edge length of 0, 1, 2, 3, and 4.
    * The edge lengths are the IDs for those
    * triangular numbers. Only one instance should be
made for each ID.
    * Class TriangleNumber represents a single triangular
number.
    */

public class TriangleNumber
{

    private final int tn; // which triangle number

    /**
     * This default constructor makes instance zero of
class TriangleNumber.
     */
    TriangleNumber()
    {
        tn = 0;
    }

    /**
     * This fully parametrized constructor makes
instance n of class TriangleNumber.
     *
     * @param    n    the ID for an instance of
TriangleNumber of size n
    */
}
```

```
    */
    public TriangleNumber(int n)
    {
        tn = n;
    }

    /**
     * Returns the ID for this instance of
    TriangleNumber.
     *
     * @return      returns the ID of this instance of
    TriangleNumber
     */
    public int getID()
    {
        return tn;
    }

    /**
     * This pseudo accessor calculates the size of this
    instance of
     * class TriangleNumber according to the equation
    found in
     * http://en.wikipedia.org/wiki/Figurate\_number.
     *
     * @return      the value or size of the triangle
     */
    public int getValue()
    {
        return (tn * (tn + 1)) / 2;
    }

    /**
```

* The gnomon is how much must be added to produce the

* next triangular number. This is a pseudo-accessor.

```

    * @return gnomon
    */
    public int getGnomon()
    {
        return tn + 1;
    }

    /**
     * Overrides toString method to display tn: value
     *
     * @return returns String tn: size
     */
    public String toString()
    {
        return "t" + getID() + ": " + getValue();
    }
}

```

The test class creates one instance of a triangular number and displays its size.

```

class TestOneTriangleNumber
{

    public static void main(String[] args)
    {
        TriangleNumber t = new TriangleNumber(6);
        System.out.println("t6: " + t.getValue());
    }
}

```

```

    }

}

```

The resulting output is **t6: 21**.

Using the instance's **toString** method produces a shorter program that is easier to understand. The **print** and **println** commands automatically call the **toString** method of an object. If the instance of an object is named **t** then the print command would be **println(t);**. Using **toString** methods allow your computer program to be briefer and easier to comprehend.

```

class TestOneTriangleNumber
{

    public static void main(String[] args)
    {
        TriangleNumber t = new TriangleNumber(6);
        // System.out.println("t6: " + t.getValue());
        System.out.println(t);
    }
}

```

The resulting output continues to be **t6: 21**.

**Topic: JOptionPane's static method
showInputDialog**

```

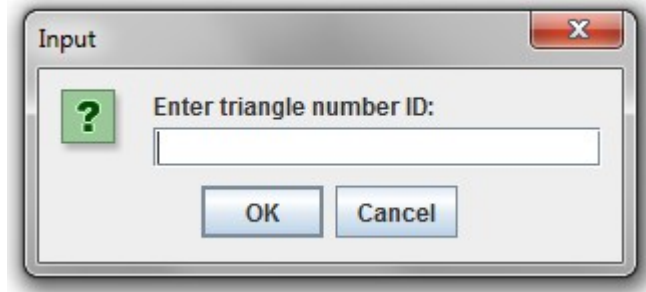
import javax.swing.JOptionPane;

```

The **showInputDialog** static method of class **JOptionPane** displays a message in a dialog box and accepts an input string in response. The **showInputdialog** method returns the user's input as an instance of **String**. Next the program converts that string to an **int** for use in its **getValue** method. Again the program uses a static method of a class. The static method **parseInt** of class **Integer** returns an **int** value matching the contents of a string. Thus **Integer.parseInt("36")** returns the **int** value of **36** as expected.

```
import javax.swing.JOptionPane;

class TestOneTriangleNumberV2
{
    public static void main(String[] args)
    {
        String triangleID =
JOptionPane.showInputDialog("Enter
triangle number ID: ");
        System.out.println("input: " + triangleID);
        int tn = Integer.parseInt(triangleID);
        TriangleNumber t = new TriangleNumber(tn);
        System.out.println(t);
    }
}
```



Version 2 of our test program produced this result.

input: 4

t4: 10

Topic: for loop

The next challenge is to list the first sixteen triangular numbers: 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, and 120. The program uses a for loop to iterate from zero to fifteen, not sixteen. The stop value is called a sentinel value. For more information go to

http://en.wikipedia.org/wiki/Sentinel_value.

```
final int STOP = 16;
for (int i = 0; i < STOP; i++)
{
    ... statements to be executed repeatedly
}
```

The for loop's control section has three portions: initialize, check for termination, and increment.

```
for (initialize; terminate; increment)
{
```

```

    ... statements to be executed repeatedly
}

```

For initialization this for loop defines a control variable named `i` and initializes its value to 0. The increment portion uses the command `i++` which is equivalent to `i = i + 1;`. That is not a mathematical statement. The `=` symbol assigns the value from the right side to the variable on the left. Thus `i + 1` is evaluated first. For example `i` begins with a value of zero, then `i + 1` has the value of 1. That value of 1 is assigned to the variable on the left side of the `=` operator. That results in the variable `i` having a new value of 1. The termination portion compares the variable `i` with the stop value of 16. Previously the program initialized an int variable named `STOP` to 16. Because that value will never change that variable was made final, thus the name is in all capital letters as `STOP`. Such variables are known as sentinel variables.

For each iteration of the loop this program makes an instance of the next triangular number and prints its contents.

```

class ListTriangleNumbers
{

    /*
    OEIS: On-Line Encyclopedia of Integer Sequences
    The sequence of triangular numbers (sequence
    A000217 in OEIS),
    starting at the 0th triangular number, is:

```

```

    0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78,
91, 105, 120
`    */
    public static void main(String[] args)
    {
        final int STOP = 16;
        for (int i = 0; i < STOP; i++)
        {
            TriangleNumber t = new
TriangleNumber(i);
            System.out.print(t.getValue() + " ");
        }
        System.out.println();
    }
}

```

The program prints this result: **0 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120** on one line because the print statement used the **print** method. Only after the for loop was completed did the program execute a **println** for a new line.

Java classes can represent mathematical objects such as a square. For more information concerning squares you can consult Wikipedia at <http://en.wikipedia.org/wiki/Square>. All squares are similar. They vary only in the length of the side. Thus the class for a square retains only the length of the side.

When a class describes an object, that class has a

name representing the object. A good name for such a class would be "Square." Thus the class's code begins as follows. Remember a class's name begins with a capital letter.

```
/**
 * Each instance of this class represents a square
 * number.
 * They are 0, 4, 9, 16, etc.
 */
public class SquareNumber
{

    private final int sqN; // square's ID

    /**
     * This default constructor creates an instance of
     SquareNumber with ID of 0.
     */
    public SquareNumber()
    {
        sqN = 0;
    }

    /**
     * This fully parametrized constructor creates an
     instance of
     * SquareNumber with ID of n.
     */
    public SquareNumber(int n)
    {
        sqN = n;
    }
}
```

```

/**
 * This accessor returns the instance's ID.
 *
 * @return the square number's ID
 */
public int getID()
{
    return sqN;
}

/**
 * This virtual accessor returns the size of the
square number.
 *
 * @return the size of the square number
 */
public int getValue()
{
    return sqN * sqN;
}

/**
 * This virtual accessor returns the additional
amount needed
 * to increase to the next square number. This is
the edge across
 * the top and on the right side of the current
square. Thus
 * the gnomon is  $2*n + 1$ .
 *
 * @return gnomon for this square
 */

```

```

public int getGnomon()
{
    return (2 * sqN) + 1;
}

}

```

Class **SquareNumber** is similar to class **TriangleNumber**. Likewise the program to list the first sixteen square numbers beginning with zero is similar to the program to list the first sixteen triangular numbers.

```

class ListSquareNumbers
{

    /*
    OEIS: On-Line Encyclopedia of Integer Sequences
    The squares (sequence A000290 in OEIS) smaller
    than 602 are:

    0**2 = 0
    1**2 = 1
    2**2 = 4
    3**2 = 9
    4**2 = 16
    5**2 = 25
    6**2 = 36
    7**2 = 49
    8**2 = 64
    9**2 = 81`
    */
    public static void main(String[] args)
    {

```

```

    final int STOP = 16;
    for (int i = 0; i < STOP; i++)
    {
        SquareNumber sq = new SquareNumber(i);
        System.out.print(sq.getValue() + " ");
    }
    System.out.println();
}
}

```

That program produces these results: "0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225" which matches sequence **A000290** in OEIS.

Topic: Interfaces

Can a program list the value or gnomon of any triangular number or square number that equals 73? For that challenge the program loads all the instances of triangular numbers and square numbers into an **ArrayList**, but **ArrayList** is limited to one class type. To solve that conundrum the program defines an interface for polygonal numbers because both triangular numbers and square numbers are examples of figurate numbers, described at http://en.wikipedia.org/wiki/Figurate_number, which are polygonal numbers that can be described as a collection of points arranged in a regular convex polygon. For further information on polygonal numbers go to http://en.wikipedia.org/wiki/Polygonal_Numbers. That interface includes the methods common to all polygonal number classes, such as `getID`, `getSize`, and `getGnomon`.

An interface sets expectations for any class implementing that interface. The interface can contain fixed variables and signatures of methods. Any class implementing that interface can use those fixed variables and must have a full method implementing the expected methods.

```
// Polygonal Numbers
```

```
public interface PolygonalNumber
{
    public int getID();
    public int getValue();
    public int getGnomon();
}
```

Both `TriangleNumber` and `SquareNumber` already have those expected methods. When the class header includes **`implements PolygonalNumber`** then instances of that class can be included in the array list defined by

```
ArrayList<PolygonalNumber> polygonalNumbers = new
ArrayList<>();
```

Class `TriangleNumber` now begins with **`public class TriangleNumber implements PolygonalNumber`**.

```
/**
 * Triangular numbers can be represented by a group of
 * dots
 * organized in an equilateral triangle.
 * The first few triangular numbers are of size 0, 1,
 * 3,
```

```
* 6, and 10. Those are the sizes of
* triangles with edge length of 0, 1, 2, 3, and 4.
* The edge lengths are the IDs for those
* triangular numbers. Only one instance should be
made for each ID.

* Class TriangleNumber represents a single triangular
number.
*/
public class TriangleNumber implements PolygonalNumber
{

    private final int tn; // which triangle number

    /**
     * This default constructor makes instance zero of
class TriangleNumber.
     */
    TriangleNumber()
    {
        tn = 0;
    }

    /**
     * This fully parametrized constructor makes
instance n of class TriangleNumber.
     *
     * @param    n    the ID for an instance of
TriangleNumber of size n
     */
    public TriangleNumber(int n)
    {
        tn = n;
    }
}
```

```
/**
 * Returns the ID for this instance of
TriangleNumber.
 *
 * @return      returns the ID of this instance of
TriangleNumber
 */
public int getID()
{
    return tn;
}

/**
 * This pseudo accessor calculates the size of this
instance of
 * class TriangleNumber according to the equation
found in
 * http://en.wikipedia.org/wiki/Figurate\_number.
 *
 * @return      the value or size of the triangle
 */
public int getValue()
{
    return (tn * (tn + 1)) / 2;
}

/**
 * The gnomon is how much must be added to produce
the
 * next triangular number. This is a pseudo-
accessor.
 * @return gnomon
```

```

    */
    public int getGnomon()
    {
        return tn + 1;
    }

    /**
     * Overrides toString method to display tn: value
     *
     * @return returns String tn: size
     */
    public String toString()
    {
        return "t" + getID() + ": " + getValue();
    }
}

```

In a similar manner class **SquareNumber** begins with `public class SquareNumber implements PolygonalNumber`.

```

    /**
     * Each instance of this class represents a square
     * number.
     * They are 0, 4, 9, 16, etc.
     */
    public class SquareNumber implements PolygonalNumber
    {

        private final int sqN; // square's ID

        /**
         * This default constructor creates an instance of

```

SquareNumber with ID of 0.

```
    */
    public SquareNumber()
    {
        sqN = 0;
    }

    /**
     * This fully parametrized constructor creates an
instance of
     * SquareNumber with ID of n.
     */
    public SquareNumber(int n)
    {
        sqN = n;
    }

    /**
     * This accessor returns the instance's ID.
     *
     * @return the square number's ID
     */
    public int getID()
    {
        return sqN;
    }

    /**
     * This virtual accessor returns the size of the
square number.
     *
     * @return the size of the square number
     */
```

```

public int getValue()
{
    return sqN * sqN;
}

/**
 * This virtual accessor returns the additional
amount needed
 * to increase to the next square number. This is
the edge across
 * the top and on the right side of the current
square. Thus
 * the gnomon is 2*n + 1.
 *
 * @return gnomon for this square
 */
public int getGnomon()
{
    return (2 * sqN) + 1;
}
}

```

The program must define the target number and the array list. Next the triangular numbers and square numbers are added to that array list. Finally the program searches its array list for the target number. The target number is 73. Because 73 is a fixed value the definition is **final int TARGET = 73**. Final variables have names in all capital letters.

Each square number has both a size and a gnomon. The

size is the square of the edge, so largest square needed would be 10 because 10 times 10 equals 100 which exceeds the target of 73. Also the gnomon must be considered. The gnomon for a square of edge n is $2n + 1$, so a square of size 50 would have a gnomon of 101, which exceeds the target of 73. Adding additional squares is not a problem. The program must include all the square whose size or gnomon is less than 73. For each ID less than 50 the program must create an instance of square number and add that instance to the array list. Repeatedly performing that task requires a loop. Since we know the loop will be repeated 50 times the program uses the standard for loop.

```
for (int i = 0; i < SQ_STOP; i++)
{
    SquareNumber sq = new SquareNumber(i);
    polygonalNumbers.add(sq);
}
```

Calculating the size and gnomon of a triangular number can be done, but it's slightly more complex, so the program uses a while loop. The gnomon is always less than the size, so the program sets up a loop. While the gnomon is less than the target size, the triangular number is added to the array list. Notice variable n has the triangular number's ID. Variable n begins with a value of zero. Each iteration of the loop increments n by one.

```
int n = 0;
TriangleNumber tri = new TriangleNumber(n);
int gnomon = tri.getGnomon ();
```

```

while ( gnomon < TARGET)
{
    polygonalNumbers.add(tri);
    n = n + 1;
    tri = new TriangleNumber(n);
    gnomon = tri.getGnomon();
}

```

This final step uses a for each loop to search the entire array list for any polygonal number that has either a value of gnomon of the right size.

```

for (PolygonalNumber p: polygonalNumbers)
{
    if (p.getValue() == TARGET)
    {
        System.out.print(p.getClass().getName() +
" " + p.getID() + " ");
    }
    if (p.getGnomon() == TARGET)
    {
        System.out.print(p.getClass().getName() +
" " + p.getID() + " gnomon ");
    }
}
System.out.println();

```

The entire program is shown below.

```

import java.util.ArrayList;

class FindNumberTypes
{

```

```
/*
OEIS: On-Line Encyclopedia of Integer Sequences
The squares (sequence A000290 in OEIS) smaller than
602 are:

0**2 = 0
1**2 = 1
2**2 = 4
3**2 = 9
4**2 = 16
5**2 = 25
6**2 = 36
7**2 = 49
8**2 = 64
9**2 = 81`
    */

/*
OEIS: On-Line Encyclopedia of Integer Sequences
The sequence of triangular numbers (sequence
A000217 in OEIS),
starting at the 0th triangular number, is:

0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91,
105, 120
`    */

/* For a given number list all its number types:
square, square's    gnomon, square, and square gnomon.
*/

public static void main(String[] args)
```

```

{
    final int TARGET = 73;

    ArrayList<PolygonalNumber> polygonalNumbers =
        new ArrayList<>();

    // Generate SquareNumbers with values below 100.
    // 10**2 = 100 so use a for loop interating from 0
to 9
    // A square's of edge length n has a gnomon of size
2n + 1,
    // so generate squares up to edge size 50.

    final int SQ_STOP = 50;
    for (int i = 0; i < SQ_STOP; i++)
    {
        SquareNumber sq = new SquareNumber(i);
        polygonalNumbers.add(sq);
    }

    // Generate triangular numbers with values below 73
    // The gnomon is always smaller than the size,
    // so use a while loop
    // stopping when the gnomon is greater than or
equal to 73.
    int n = 0;
    TriangleNumber tri = new TriangleNumber(n);
    int gnomon = tri.getGnomon ();
    while ( gnomon < TARGET)
    {
        polygonalNumbers.add(tri);
        n = n + 1;
        tri = new TriangleNumber(n);
    }
}

```

```

        gnomon = tri.getGnomon();
    }

    System.out.print(TARGET + ": ");
    // Use a for each loop to search the array list
    for (PolygonalNumber p: polygonalNumbers)
    {
        if (p.getValue() == TARGET)
        {
            System.out.print(p.getClass().getName() +
" " + p.getID()          + " ");
        }
        if (p.getGnomon() == TARGET)
        {
            System.out.print(p.getClass().getName() +
" " + p.getID()          + " gnomon ");
        }
    }
    System.out.println();
}
}

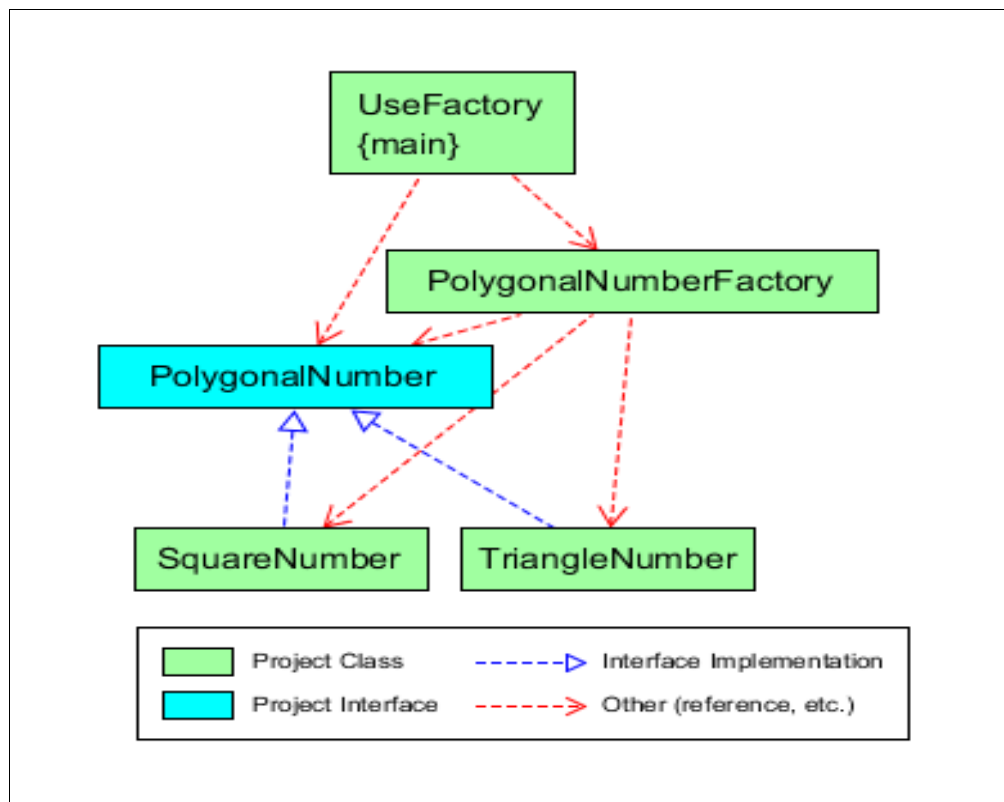
```

The program's result is **73: SquareNumber 36 gnomon**.

This program used all three loop methods: for each, while, and for. Obviously each style of loop meets the needs of different situations. A strong knowledge of loops in java is essential for a java computer language programmer.

Topic: Refactoring

The computer program works well with square and triangular numbers, but concerns arise when the program must begin processing pentagonal (5-sided), hexagon(6-sided), heptagonal(7-sided), octagonal(8-sided), nonagon(9-sided), decagonal(10-sided), and more polygonal numbers as described at http://en.wikipedia.org/wiki/Polygonal_Numbers. Each polygonal number class currently requires a separate loop to generate the polygonal numbers. Including more number classes would make the program overly complex, so the program should be re-factored. A factory method generates upon request an instance of any of the classes that implement an interface. For this computer program a factory method could produce either Square or Triangular numbers as requested by the main program. Factory methods reduce the complexity of the main method.



Now the main method of the **UseFactory** class creates an instance of **PolygonalNumber** factory. The command is **PolygonalNumberFactory pnFactory = new PolygonalNumberFactory();** Then the computer program must loop through a list of the types of polygonal numbers, so the names of those types of polygonal numbers are stored in an array. The command is **String[] polygonalNames = { "TRIANGLE", "SQUARE" };** to produce all the instances of the square and triangular polygonal numbers. For each such type of polygonal number the program creates another instance of that class with a larger ID until its value or gnomon exceed the requires. The command to create the polygonal number is **PolygonalNumber polyNum = pnFactory.createPolygonalNumber(pn, id);** A while loop implements that loop which continues creating new polygonal numbers until the requirements are satisfied. Here is the source code for that while loop.

```

    int id = 0;
    int gnomon = 0;
    int value = 0;
    while (( gnomon < TARGET ) ||
           ( value < TARGET))
    {
        PolygonalNumber polyNum =
            pnFactory.createPolygonalNumber(pn, id);
        polygonalNumbers.add(polyNum);
        id++;
        value = polyNum.getValue();
        gnomon = polyNum.getGnomon();
        // System.out.println("Add to array pn: " +
pn + " id: " +          id + " value: " + value + "

```

```
gnomon: " + gnomon);
    }
```

During each iteration of the while loop each polygonal number is added to an array list named **polygonalNumbers**. The id number must be incremented by the command **id++**; so the next iteration will advance to the next polygonal number. The value and gnomon are retrieved from the instance of the polygonal number so the while loop can test them for completion of the loop. The test command is **((gnomon < TARGET) || (value < TARGET))** so the loop continues to be iterated until both the gnomon and the value exceed the sentinel value in the constant named **TARGET**.

The main method contains this source code.

```
    final int TARGET = 73;

    ArrayList<PolygonalNumber> polygonalNumbers =
    new ArrayList<>();

    // Generate TriangleNumbers and SquareNumbers with
    // values and gnomons below 100.
    String[] polygonalNames = { "TRIANGLE", "SQUARE" };
    PolygonalNumberFactory pnFactory = new
    PolygonalNumberFactory();
    for (String pn : polygonalNames)
    {
        int id = 0;
        int gnomon = 0;
        int value = 0;
        while (( gnomon < TARGET ) ||
```

```

        ( value < TARGET))
    {
        PolygonalNumber polyNum =
pnFactory.createPolygonalNumber(pn, id);
        polygonalNumbers.add(polyNum);
        id++;
        value = polyNum.getValue();
        gnomon = polyNum.getGnomon();
        // System.out.println("Add to array pn: "
+ pn + " id: " + id + " value: " + value + " gnomon: "
+ gnomon);
    }
}

```

The for-each loop that searches through all the polygonal numbers for matches with the **TARGET** value remains the same. Observe the main method simply works with polygonal numbers. No longer is the main method concerned with creating square and triangular numbers. All that work moved to the instance of the factory class. Factory classes do make the main program easier to write and understand.

```

/**
 * Creates instances of the concrete classes that
 * implement the PolygonalNumber interface.
 */
public class PolygonalNumberFactory
{

    /**
     * This method returns an instance of a concrete
     * class that implements the PolygonalNumber interface.

```

```

*
* @return instance of concrete class implement
PolygonalNumber interface
*/
public PolygonalNumber createPolygonalNumber(String
concreteClass,      int id)
{
    PolygonalNumber pn = null;
    switch(concreteClass)
    {
        case "TRIANGLE":
            pn = new TriangleNumber(id);
            break;
        case "SQUARE":
            pn = new SquareNumber(id);
            break;
    }
    return pn;
}
}

```

The factory class uses a switch statement based on the name of the concrete class. Adding pentagonal (5-sided), hexagon(6-sided), heptagonal(7-sided), octagonal(8-sided), nonagon(9-sided), decagonal(10-sided), and more polygonal numbers requires simply to add more case statements and write the source code for those class descriptions of those types of polygonal numbers.

```

switch(concreteClass)
{
    case "TRIANGLE":

```

```

        pn = new TriangleNumber(id);
        break;
    case "SQUARE":
        pn = new SquareNumber(id);
        break;
    case "PENTAGONAL":
        pn = new PentagonalNumber(id);
        break;
    case "HEXAGON":
        pn = new HexagonNumber(id);
        break;
    case "HEPTAGONAL":
        pn = new HeptagonalNumber(id);
        break;
    }
    return pn;

```

Although the program is longer, the source code is simple to expand and remains easy to comprehend.

Topic: Compiler warning

A computer programmer might be tempted to write the factory class in a briefer fashion by immediately using a **return** statement. For example, this erroneous program demonstrates that mistake.

```

/**
 * Creates instances of the concrete classes that
 * implement the PolygonalNumber interface.
 */
public class PolygonalNumberFactoryv2
{

```

```

/**
 * This method returns an instance of a concrete
 * class that implements the PolygonalNumber
interface.
 *
 * @return instance of concrete class implement
PolygonalNumber interface
 */

public PolygonalNumber createPolygonalNumber(String
concreteClass, int id)
{
    switch(concreteClass)
    {
        case "TRIANGLE":
            return new TriangleNumber(id);
        case "SQUARE":
            return new SquareNumber(id);
    }
}
}

```

The **break** command is no longer needed because the **return** command immediately ends the **switch**. The compiler studies this source code and provides this error message.

```

PolygonalNumberFactoryv2.java:24: error: missing
return statement

```

```

    }

```

^

1 error

The compiler is uncertain that the method contains the return statement required by that method's declaration of a return type of **PolygonalNumber**.

The solution to this compiler error message is to move the **return** statement to the end of your method. When your code produces the return value earlier a variable temporarily holds the value. Then the **break** statement drops the execution down to the last statement which was the required **return** statement.

Always ending your method with the return statement means a few more commands are needed but the code is easy to understand.

Topic: Activity

Search Wikipedia's Polygonal Numbers topic for a relationship between square and triangular numbers. Write a java program to verify the first few values.

Chapter Five: java interface

Topic: Cartesian Coordinate System

Source: Wikipedia <http://en.wikipedia.org/wiki/File:Cartesian-coordinate-system.svg>

The Cartesian coordinate system describing two dimensional Euclidean space uses the real number system to label x and y coordinates crossing at a right angle on a flat plane. Each point is identified as (x,y) where x and y are the coordinates. The origin is at (0,0). For more information concerning this coordinate system go to

http://en.wikipedia.org/wiki/Cartesian_coordinates.

Source: <http://docs.oracle.com/javase/tutorial/2d/overview/coordinate.html>

Topic: Java Graphics coordinates

The base class CartesianShape describes an arbitrary shape. Only the quadrant with both positive x and positive y coordinates are considered because Java provides its own graphics system with that limitation. After construction of a system of classes describing points, line segments, triangles, rectangles, and circles, another Java program can display this computer program's results using Java's own graphic system of geometric primitives, such as point, line, triangle, rectangle, and oval. The base class CartesianShape is so theoretical that it does not describe any commonly named shape. It merely knows the Cartesian (x,y) address top left corner point. Basic methods for moving that point are included in the base class.

Topic: Abstract Class

An abstract method for area is described because some shapes, such as a point, or line, have no area. Also the computation of area depends on the specific shape, such as rectangle, square, circle, or triangle. Each such sub class of shape has its own mathematical equation for area. Nonetheless all those shapes have an area, even if might be zero. Because area is an abstract method, which means it has only a method signature, not a body, the enclosing class, CartesianShape, must also be labeled abstract. The Java command, **new**, cannot be used to make an instance, or object, of an abstract class.

```
/**
    Use only the one quadrant of the Cartesian grid
    with positive x and y coordinates. The origin (0,0) is
    at the top left corner. The x-coordinate increases
    from left to right. The y-coordinate increases from
    top to bottom. This description does not match the
    diagram above.
*/
public abstract class CartesianShape
{
    private double topLeftX;
    // X coordinate of top left corner of bounding
    rectangle
    private double topLeftY;
    // Y coordinate of top left corner of bounding
    rectangle
}
```

```
/**
    default constructor. Shape is assumed to begin at
    (0,0).
*/
public CartesianShape()
{
    topLeftX = 0;
    topLeftY = 0;
}

/**
    full constructor.
*/
public CartesianShape(double x, double y)
{
    topLeftX = x;
    topLeftY = y;
}

/**
    Return x coordinate of top left corner
    of bounding rectangle.
*/
public double getTopLeftX()
{
    return topLeftX;
}

/**
    set x coordinate of top left corner of
    bounding rectangle.
*/
public void setTopLeftX(int x)
```

```
{
    topLeftX = x;
}

/**
    Return y coordinate of top left corner
    of bounding rectangle.
*/
public double getTopLeftY()
{
    return topLeftY;
}

/**
    set y coordinate of top left corner of
    bounding rectangle.
*/
public void setTopLeftY(double y)
{
    topLeftY = y;
}

/**
    Increment x and y coordinates of top left
    corner
    of bounding rectangle.
*/
public void move(double x, double y)
{
    topLeftX += x;
    topLeftY += y;
}
```

```
/**
    Change x and y coordinates of top left corner
    of bounding rectangle.
*/
public void moveTo(double x, double y)
{
    topLeftX = x;
    topLeftY = y;
}

/**
    Return area of shape. Zero if not closed
loop..
*/
public abstract double getArea();
}
```

Unlike normal practice no test class accompanies this abstract class. The Java program cannot use the command, new, to make an instance, or object, of an abstract class. Therefore no test class was written to test abstract class CartesianShape.

Another class can extend an abstract class. When all the abstract methods are described, thus having a body of statements for each method, then that new child class is not abstract.

Class CartesianRectangle extends abstract class CartesianShape and provides a full definition for the area method, thus class CartesianRectangle is not an abstract class. CartesianShape is the parent and

`CartesianRectangle` is the child class in that relationship. A rectangle is defined by its length and width, so `CartesianRectangle` has length and width as attributes. Naturally constructors, accessors, and mutators for each attribute are included.

```
/**
    Extends CartesianShape to define a Cartesian grid
    based rectangle.

    The rectangle is always oriented along the x and y
    axes.
*/
public class CartesianRectangle extends CartesianShape
{

    private double length;
    private double width;
    // Length and width are never negative values.

    /**
        Default constructor
    */
    public CartesianRectangle()
    {
        length = 0;
        width = 0;
    }

    /**
        Fully parametrized constructor
    */
    public CartesianRectangle(double x, double y,
        double len, double wide)
```

```
{
    super(x,y);
    if (len >= 0 && wide >= 0)
    {
        length = len;
        width = wide;
    }
}

/**
    Accessor for attribute length
*/
public double getLength()
{
    return length;
}

/**
    Mutator for attribute length
*/
public void setLength(double len)
{
    if (len >= 0)
    {
        length = len;
    }
}

/**
    Accessor for attribute width
*/
public double getWidth()
{
```

```
        return width;
    }

    /**
     * Mutator for attribute width
     */
    public void setWidth(double wide)
    {
        if (wide >= 0)
        {
            length = wide;
        }
    }

    /**
     * Defines the abstract method of
     * CartesianShape.
     * GetArea acts likes an accessor, so
     * area is a pseudo-attribute.
     */
    public double getArea()
    {
        return length * width;
    }
}
```

The test class sets up a CartesianRectangle and displays its data. A second such rectangle is moved to another location and its information is printed.

```
public class TestCartesianRectangle
{
```

```

public static void main(String[] args)
{
    CartesianRectangle rect = new
CartesianRectangle();
    System.out.println("rectangle 1 x: " +
rect.getTopLeftX()
        + " y: " + rect.getTopLeftY()
        + " length: " + rect.getLength() + "
width: " + rect.getWidth() + " area:
" + rect.getArea());
    CartesianRectangle rect2 =
        new CartesianRectangle(5, 3, 20.5,
3.09);
    System.out.println("rectangle 2 x: "
        + rect2.getTopLeftX() + " y: " +
rect2.getTopLeftY()
        + " length: " + rect2.getLength() + "
width: " + rect2.getWidth() + "
area: " + rect2.getArea());
    rect2.moveTo(100,200);
    System.out.println("rectangle 2 moved to
(100,200).");
    System.out.println("rectangle 2 x: "
        + rect2.getTopLeftX() + " y: " +
rect2.getTopLeftY()
        + " length: " + rect2.getLength() + "
width: "
        + rect2.getWidth() + " area: " +
rect2.getArea())
    }
}

```

The test class produces this result. The default constructor produced the first rectangle so the location of the top left corner is (0,0), the length is zero, and the width is zero as expected from a default constructor. The second rectangle was set at (5,3) which was passed to the constructor of the CartesianShape class. Also the length was set to 20.5 and width to 3.09. Area is a pseudo-attribute resulting from the multiplication of length by width.

```
rectangle 1 x: 0.0 y: 0.0 length: 0.0 width: 0.0
area: 0.0

rectangle 2 x: 5.0 y: 3.0 length: 20.5 width: 3.09
area: 63.345

rectangle 2 moved to (100,200).

rectangle 2 x: 100.0 y: 200.0 length: 20.5 width:
3.09 area: 63.345
```

Not only does CartesianRectangle extend CartesianShape, both of those classes extend class Object. In fact all Java classes extend class Object.

java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass.

Source:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

Class `Object` provides several methods including `toString`.

<code>String</code>	<code>toString()</code> Returns a string representation of the object.
---------------------	---------------------------------------------------------------------------

Thus our test program can be simplified. Instead of a long print command, the main method can simply use `System.out.println(rectangle);`

```
public class Test2CartesianRectangle
{

    public static void main(String[] args)
    {
        CartesianRectangle rect = new
CartesianRectangle();
        System.out.println(rect);
        CartesianRectangle rect2 =
            new CartesianRectangle(5, 3, 20.5, 3.09);
        System.out.println(rect2);
        rect2.moveTo(100,200);
        System.out.println("rectangle 2 moved to
(100,200).");
        System.out.println(rect2);

    }
}
```

The new simplified test class prints this cryptic report.

CartesianRectangle@1cd8f55c**CartesianRectangle@67d479cf****rectangle 2 moved to (100,200).****CartesianRectangle@67d479cf**

The `toString` method of class `Object` simply prints the object's class name and address in memory. Naturally such reports do not satisfy the user's needs to know the location of each rectangle, nor their sizes. Thankfully inherited methods can be overridden. The child class simply includes a new definition for that method. The method's return type and name must be the same. For example, a new definition for `toString` is shown below.

```

/**
    Override default toString method inherited from
    object named Object.
 */
public String toString()
{
    return "rectangle at (" + getTopLeftX() + ","
        + getTopLeftY() + ") length: " + length + " width: "
        + width + " area: " + getArea();
}

```

Now the new test class displays this more informative report.

```

rectangle at (0.0,0.0) length: 0.0 width: 0.0 area: 0.0
rectangle at (5.0,3.0) length: 20.5 width: 3.09 area: 63.345
rectangle 2 moved to (100,200).
rectangle at (100.0,200.0) length: 20.5 width: 3.09 area:
63.345

```

Many computer programmers routinely override the `toString` method for every class, so testing work will be simple.

Topic: Using Java graphics to study infinite series



Source: <http://xkcd.com/994/>

A most famous infinite series is from Zeno's paradox. The series can be represented in modern mathematical symbols as:

$$\sum_{n=1}^{\infty} \frac{1}{2^n} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

That series should equal 1. More information can be found in Wikipedia at

http://en.wikipedia.org/wiki/Zeno%27s_paradoxes.

Topic: GUI

A graphical user interface (GUI) is a collection of Java classes supporting the display of images, text, and drawings on a two dimensional computer screen with the possibility of users interacting with that screen either through touch or mouse input.

<http://en.wikipedia.org/wiki/GUI> describes GUI interfaces in further detail. The original Java classes, contained in the Abstract Windows Toolkit (AWT), are also known as heavyweight components because of their dependency upon the native methods of the local machine. The AWT is being replaced by the Swing component classes, which mirror their AWT classes, but are named lightweight components because they do not depend on the machine's native GUI resources. As a result Java has both a AWT Frame class and a corresponding Swing **JFrame** class. Because AWT is being phased out, all new programming should use Swing classes. The Swing components can be classified into three main groups: component classes, container classes, and helper classes. Examples of the component classes include **JButton**, **JLabel**, and **JTextField** which define buttons, labels, and text fields. Examples of container classes include **JFrame**, **JPanel**, and **JApplet**. Applets are used with Web pages. Frames are used with other Java programs. Panels are sub-containers that can be used within a frame or an applet. Examples of the helper classes are Graphics, Color, and Font. We

will use those classes in the following example programs.

Topic: JFrame

The first step is to produce a plain frame within a window. A frame acts like a small window, so the variable is named window. The Java command **JFrame window = new JFrame();** creates a new frame named window.

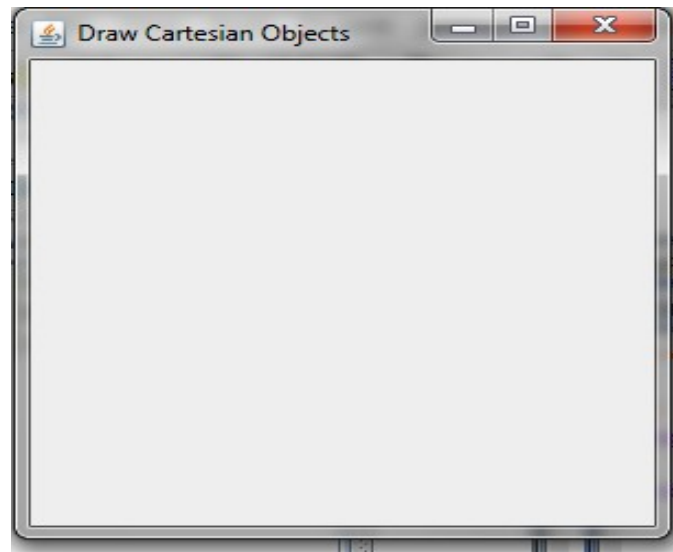
```
/**
 * Use JFrame for a GUI program
 */
import javax.swing.JFrame;

public class DrawCartesianObjects {

    public static void main(String[] a) {
        JFrame window = new JFrame();

        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        int width = 300;
        int length = 300;
        window.setBounds(30, 30, width, length);
        window.setTitle("Draw Cartesian Objects");
        window.setVisible(true);
    }
}
```

That program displays this window.

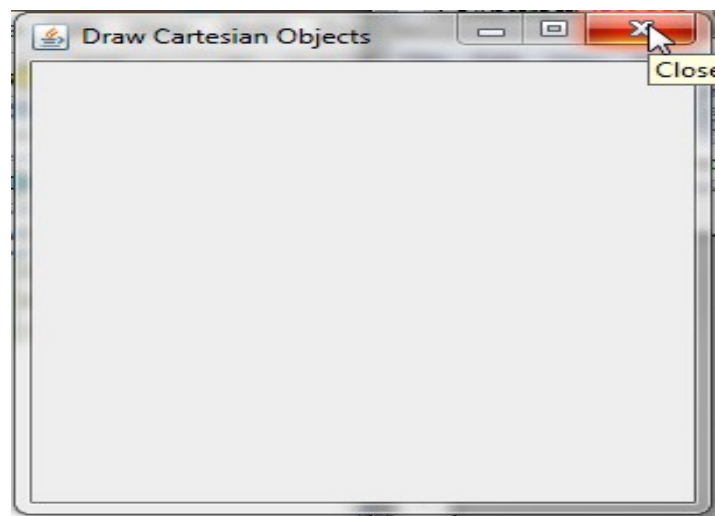


Because of the command

```
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

pressing the normal close box will close this window.

Forgetting that command means pressing the close box will not close the window.

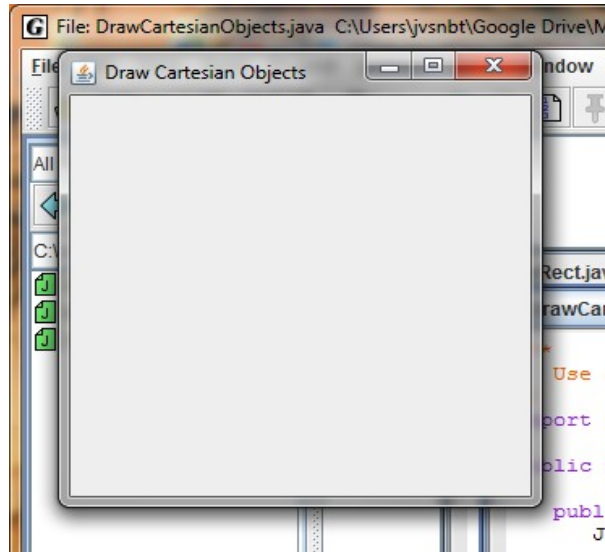


The commands

```
int width = 300;  
int length = 300;  
window.setBounds(30, 30, width, length);
```

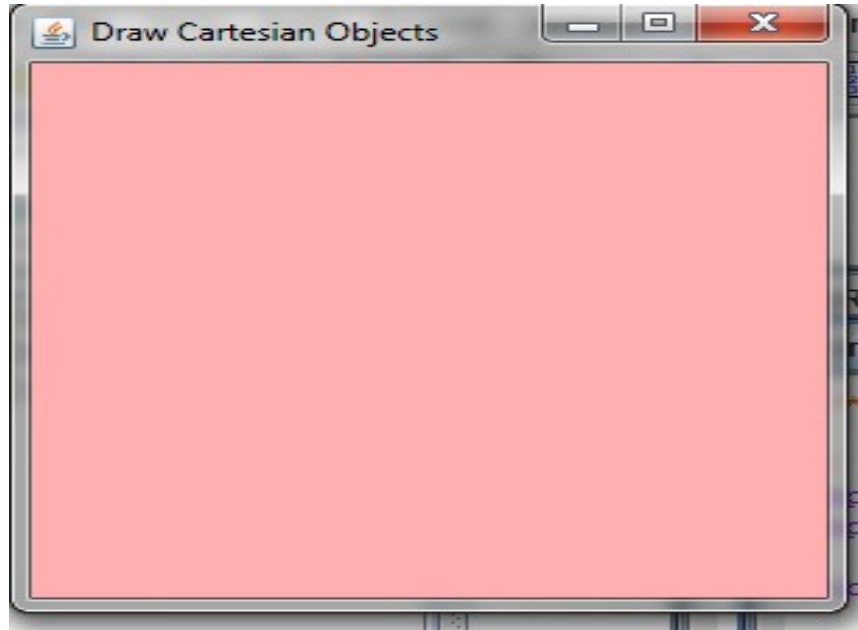
set the top left corner of the frame at point (30, 30), the width to 300 pixels and the length to 300 pixels as shown below. Notice that

window.setTitle("Draw Cartesian Objects"); set the title of the new frame.



Suppose you want to change the background color to pink. Use the command

window.getContentPane().setBackground(Color.PINK); A **JFrame** has both a content pane and a menu bar, **JMenuBar**. Buttons, labels, and text fields are automatically added to the content pane. To change the content pane, use the method **getContentPane**. To reduce lines of code the method **setBackground** is applied to the result of **window.getContentPane()**, thus one command completes several actions. **PINK** is a static value within the **Color** class, so the color is described as **Color.PINK**.



Instead of the name **window**, many programmers would use the name **frame**.

Next the program adds an object of class `JComponent` to the frame. `JComponent` is an abstract class that is a base class for all Swing components except top-level containers. Thus our class named `MyCanvas` must extend `JComponent`. Instances of `MyCanvas` must be added to a top level container, such as `JFrame`, `JDialog`, or `JApplet`. Those top-level Swing containers provide a place for other Swing components to paint themselves. Because class `MyCanvas` is of interest only to this program that class is included within our java program. Notice class `MyCanvas` lacks the attribute **public**.

```
class MyCanvas extends JComponent {  
  
    protected void paintComponent(Graphics g) {  
        Graphics2D g2 = (Graphics2D) g;
```

```

    int x = 10;
    int y = 10;
    int width = 200;
    int height = 200;
    // Draw outline of rectangle
    Rectangle surround = new Rectangle(x, y,
width, height);
    g2.draw(surround);
}
}

```

Our program's main method will create a new instance of `MyCanvas`. When that instance is added to the frame or when it is repainted the Swing toolkit automatically creates a graphics objects of type `Graphics2D`, which is an extension of the basic class **Graphics** and passes that to the `paintComponent` method. **Graphics** is a more primitive class than `Graphics2D`, so the first command casts that object as an instance of **Graphics2D**. Now the method can use the more advanced methods, such as `draw` and `fill`, which automatically draw any object implementing the `Shape` interface. Our `CartesianRectangle` class does not yet implement the **Shape** interface because that would involve considerable effort, so our current method creates an object of class `Rectangle` which does satisfy the **Shape** interface. That new rectangle is named `surround` because it will contain the planned series of rectangles representing Zeno's series of one-half, one-fourth, etc.

Topic: Anonymous Objects

The main method will must create an instance of **MyCanvas** and attach it to the instance of **JFrame** named window. Until now all programs have declared a new variable, so the main method would have included this command, **MyCanvas canvas = new MyCanvas();** Next, the program would get the content pane for the **JFrame** named window, so the next command would have been **Container content = window.getContentPane();** Finally the program would add the canvas to the content pane with the command **content.add(canvas);** As a convenience java allows programs to add items directly to the frame without fetching the content pane, so the last two commands can be condensed as **window.add(canvas);** This abbreviation is permitted only with the add method. Recall that **getContentPane** was necessary to change the background color of the content pane of a frame.

Java also allows the use of anonymous items. The canvas object will only be used once in this program when it is added to window, the **JFrame** object, so no name is needed. In such a situation programmers use anonymous variables which occur when you create an object and immediately use it. Thus the three commands are reduced to simply **window.add(new MyCanvas());** The new canvas has no name, thus it is anonymous. The use of anonymous instances can greatly abbreviate your program, reduce complexity, and make your program easier to comprehend.

```
import java.awt.*;
import java.awt.Graphics;
import java.awt.Rectangle;
```

```
import javax.swing.JComponent;
import javax.swing.JFrame;

class MyCanvas extends JComponent {

    protected void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        int x = 10;
        int y = 10;
        int width = 200;
        int height = 200;
        // Draw outline of rectangle
        Rectangle surround = new Rectangle(x, y,
width, height);
        g2.draw(surround);
    }
}

public class DrawRectSeries2 {

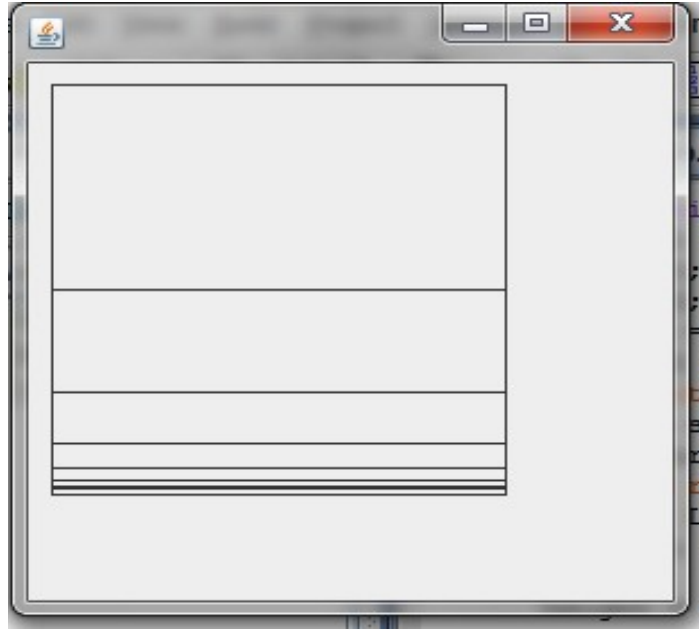
    public static void main(String[] a) {
        JFrame window = new JFrame();

window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        int width = 300;
        int length = 300;
        window.setBounds(30, 30, width, length);
        window.add(new MyCanvas());
        window.setVisible(true);
    }
}
```

Next expand the **paintComponent** method to draw a series of nine stacked rectangles of heights $\frac{1}{2}$, $\frac{1}{4}$, etc. Include the following code.

```
// draw 9 rectangles of length 1/2, 1/4, etc
final int LIMIT = 9;
for (int i = 0; i < LIMIT; i++)
{
    height = height / 2;
    Rectangle box = new Rectangle(x,y,width,height);
    g2.draw(box);
    y = y + height;
}
```

Because the number of rectangles is known a for loop was chosen, not a while loop, nor a for-each loop. The limit of 9 is a constant thus it is defined with the attribute of final and the variable's name is in all capital letters. For each rectangle we reduce its height by one-half. The program then creates a new Rectangle of that size. Because class Rectangle satisfies interface Shape, the Graphics2D method draw can draw that rectangle. The final step is to adjust the top-left point (x,y) downwards, so that point (x,y) will become the top-left point of the next rectangle. The newly expanded method produced this result.



The addition of fill color for each rectangle would improve our understanding of the situation. During the for loop the counter varies from 0 to 9. By alternating colors from orange to black based on whether that loop counter is even or odd, the program can produce alternating colors. The `%` operator returns the remainder of an integer division. "`i % 2`" returns either 0 or 1. The use of an if statement would be as shown here.

```
Color color = Color.black;
...
if (i == 0)
{
    color = Color.ORANGE;
}
else
{
    color = Color.BLACK;
}
```

```
g2.setColor(color);
```

Programmers often prefer the **switch** statement because it is so easy to read and comprehend. The **case** condition must be an integer, a long integer, a **byte** character, such as 'a', or a **String**. When a case condition matches the control variable, which is an anonymous variable holding the result of "**i % 2**" the commands following the colon are executed. The **break** command is essential, or all the commands in all following cases will be executed also. The **break** command exits control from the **switch** statement. **Break** commands are also used to exit from a loop.

```
// draw 9 rectangles of length 1/2, 1/4, etc
// with alternating colors
final int LIMIT = 9;
Color color = Color.BLACK;
for (int i = 0; i < LIMIT; i++)
{
    switch (i % 2)
    {
        case 0: color = Color.ORANGE;
                break;
        case 1: color = Color.BLACK;
                break;
    }
    g2.setColor(color);
    height = height / 2;
    Rectangle box = new Rectangle(x,y,width,height);
    g2.fill(box);
    y = y + height;
```

```
}
```

The improved paint method now produces alternating orange and black rectangles almost filling the entire box. Thus the programmer can see that in this one instance Zeno's series approaches one.



Topic: Activity

Write a java computer program to demonstrate the correctness of other infinite series for the first few iterations.

How can your knowledge of java computer programming help you find more relationships in the field of polygonal numbers?

Chapter Six: java interface

Topic: array

An array is a fixed-size collection of variables all of the same type. The declaration **int prime** declares a variable with the name **prime** and the type of **int**. The symbol **[]** following a primitive or class type changes the declaration from a single variable to an array of variables of that same type. The declaration **int[] primes** declares an array named **primes** containing a collection of variables all of type **int**. The command **int[] primes = new int[10];** both declares variable **primes** as the name of an array of variables all of type **int** and creates a new array of ten variables. Those variables are addressed from 0 to 9. The first variable is identified as **primes[0]**. The last variable is **primes[9]**. The Java Virtual Machine automatically checks the range of your index, so if your java program attempts to access the contents of array[10] , array[-1], or worse your program will receive an error message.

All those variables in array **primes** are still empty. You could painstakingly enter each prime number one at a time by using commands such as **primes[0] = 1;** or **primes[4] = 7;** Thankfully java provides us an easier way to initialize all those variables using an array initializer. When the array is declared equal to a list the array's length is automatically set to the length of that list and the values are all set up in the array. For example

```

    int[] primes = { 1, 2, 3, 5, 7, 9, 11, 13, 17, 19,
23, 25,
        29, 31, 33, 35, 37, 41, 43, 47, 49, 53, 59, 61,
65,
        67, 69, 71, 73, 77, 79, 83, 89, 91, 97, 101,
103, 107,
        109, 113, 125, 127, 129, 131, 133, 137, 139,
141, 143,
        145, 149, 151, 155, 157, 161, 163, 167, 169,
173, 179, 181 };

```

will declare a variable named **primes** as an array of variables all of type **int**. Also the values will be set up in that array. That single command has the same result as the following list of commands.

```

int[] primes = new int[61];
primes[1] = 1;
...
primes[61] = 181;

```

The array declaration and array initializer must be used in one command. You cannot use two commands. The following sequence of commands will produce an error message.

```

int[] primes;
primes = { 1, 2, 3, 5, 7, 9, 11, 13, 17, 19, 23,
25,
        29, 31, 33, 35, 37, 41, 43, 47, 49, 53, 59, 61,
65,
        67, 69, 71, 73, 77, 79, 83, 89, 91, 97, 101,

```

```

103, 107,
    109, 113, 125, 127, 129, 131, 133, 137, 139,
141, 143,
    145, 149, 151, 155, 157, 161, 163, 167, 169,
173, 179, 181 };

```

An array differs from an **ArrayList** and the object named **Array**. Also an array is not a member of the java collections framework. With an **ArrayList** object the **add** method will always successfully add another item to the array list. There is no **add** method for an array. Once the size of an array has been set the array size cannot be changed.

The following class sets up an array of int values containing all the prime numbers from 1 to 181. It has been known that prime numbers are infinite in size since Euclid offered proof in his Elements that there is no largest prime number.

(http://en.wikipedia.org/wiki/Euclid%27s_theorem)



OEIS provides access to an encyclopedia of integer sequences. Besides stimulating our imagination, these integer sequences allow us to check the correctness of our java computer

program's results. Each sequence has its own internet address. For example the prime numbers are listed at <http://oeis.org/A090332>

```
    This class works with the prime numbers from 1
through 181.
*/
public class Primes
{
    // An array of int primitive values, not object
    instances of classes
    // List of prime numbers. Source:
http://oeis.org/A090332
    private int[] primes = { 1, 2, 3, 5, 7, 9, 11, 13,
17, 19, 23, 25,
        29, 31, 33, 35, 37, 41, 43, 47, 49, 53, 59, 61,
65,
        67, 69, 71, 73, 77, 79, 83, 89, 91, 97, 101,
103, 107,
        109, 113, 125, 127, 129, 131, 133, 137, 139,
141, 143,
        145, 149, 151, 155, 157, 161, 163, 167, 169,
173, 179, 181 };

    /**
     default constructor
    */
    public Primes()
    {

    }

    /*
     This method works with the prime numbers.
    */
    private void go()
    {
```

```
        System.out.println("I know these prime
numbers:");
        for (int prime : primes)          // for each grade
in grades
            System.out.println(prime);    // print that
grade
    }

    /**
     * main method
     * obj is a common abbreviation of "object".
     */
    public static void main(String[] args)
    {
        Primes obj = new Primes();
        obj.go();
    }
}
```

The program produces a list of prime numbers.

Program Results

The program produces a list of prime numbers from 1 to 181.

I know these prime numbers:

```
1
2
3
...
181
```

The list has been abbreviated with an ellipsis (...) replacing a large stretch of the listing.

Notice that the default constructor has no commands. Removing that empty default constructor would shorten your program and still produce the same results. Commonly an IDE automatically adds such empty default constructors. Programmers usually leave those empty default constructors in their programs so that later they can easily expand the empty constructor. Of all the new programming languages java has the largest amount of comments, requires everything be defined before being used, and generally has the largest source files.

Often for a short quick development a programmer will combine the test class and the object class into one class as shown above. But serious development requires more diligence at following the rules of object-oriented programming. For a more object-oriented approach to this program, the prior class has been divided into two classes: a class named **Primes** which maintains a list of prime numbers and a separate class named **TestPrimes** to exercise class **Primes**.

Topic: Class Primes

The class named **Primes** continues to use the first few prime numbers from the integer sequence A090332 of oeis.org to initialize an array named **primes**. In java names are case sensitive so there is no confusion, at least to the compiler, between the array **primes** and

the class **Primes**. The array's name begins with a lowercase letter p and the class' name begins with a capital letter P. Normally programmers avoid using such similar names differing only by starting with a capital letter. A better idea would be to name the class **PrimeNumbers**, not **Primes**, then those names of the array and class would not be so similar. Notice below that the class name is **PrimeNumbers**, not **Primes**. The class includes the method **isPrime** for deciding whether an integer is a prime number, **getPrime** for getting the prime number from a location in the array, and **getMaxPrime** returns the largest prime number known to this class.

```
/**
    This class works with the prime numbers from 1
    through 181.
 */
public class PrimeNumbers
{
    // An array of int primitive values, not object
    instances of classes
    // List of prime numbers. Source:
    http://oeis.org/A090332
    private int[] primes = { 1, 2, 3, 5, 7, 9, 11, 13,
17, 19, 23, 25,
        29, 31, 33, 35, 37, 41, 43, 47, 49, 53, 59, 61,
65,
        67, 69, 71, 73, 77, 79, 83, 89, 91, 97, 101,
103, 107,
        109, 113, 125, 127, 129, 131, 133, 137, 139,
141, 143,
        145, 149, 151, 155, 157, 161, 163, 167, 169,
```

```
173, 179, 181 };
```

```
/**
    default constructor
*/
public PrimeNumbers()
{

}

/**
    Test if an integer is a prime.
*/
public boolean isPrime(int n)
{
    boolean answer = false;
    for (int prime : primes)        // foreach prime
in primes
        if (n == prime)    // test prime
        {
            answer = true;
            break; // interrupt loop
        }
    return answer;
}

/**
    * Return prime from array.
*/
public int getPrime(int index)
{
    return primes[index];
}
```

```

    /**
     * Return largest prime from array, which is the
    last entry.
    */
    public int getMaxPrime()
    {
        return primes[primes.length-1];
    }

    /**
     * Return prime from array.
    */
    public int getMaxCntPrimes()
    {
        return primes.length;
    }
}

```

Topic: Scope

The array named **primes** is an attribute of the class, so it is declared before any constructor or method. All the methods have access to the array named **primes** and can use that variable in their commands. The array named **primes** is not limited to any method thus the array named **primes** has a scope of the entire class. Normally a variable's scope is as limited as possible. In this class all the methods need access to the variable named **primes**, so that variable has class scope. Variables at the start of the class with class scope and containing information that is the main goal

of the class are called attributes in object-oriented terminology.

Method **getMaxCntPrimes** simply returns the length of the array named **primes**. Each array has an associated length. In our example the array named **primes** knows its own length which the program can obtain by simply using **primes.length**.

```
/**
 * Return prime from array.
 */
public int getMaxCntPrimes()
{
    return primes.length;
}
```

Method **getPrime** returns the prime number stored at a certain location. The location of number in an array is shown by its index, which for the first location is zero. The next location is at index 1. For an array of five entries, the index varies from 0 to 4.

Notice this method does not check that the index is correct. Suppose the user asked for the prime number at index 5. That presents an impossible situation. Currently this method ignores that problem. Later versions of this method will fix this situation.

```
/**
 * Return prime from array.
 */
public int getPrime(int index)
{
```

```

    return primes[index];
}

```

Method **getMaxPrime** returns the largest prime number in the list. Since those prime number are conveniently stored in ascending numeric order, this method simply returns the last entry in the list. An array of length five has five items with addresses from 0 through 4. The addresses of array entries begin with zero, not one. So the last entry in **primes** is at location **primes.length - 1**. Thus the method returns **primes[primes.length - 1]**. For an array of length five **primes.length** would be five, so **primes.length - 1** would be four, which is the address of the last entry. Then **return primes[primes.length - 1];** is equivalent to **return primes[4];** When those numbers are stored in ascending sequence **return primes[4]** will return the largest number in that list.

```

/**
 * Return largest prime from array, which is the
 * last entry.
 */
public int getMaxPrime()
{
    return primes[primes.length - 1];
}

```

Method **isPrime** currently uses a for-each loop to search the entire array for the target value. When the target value is found the method returns a **boolean** value of **true**. Otherwise the method returns the default value of **false**. Some programmers use a **return**

statement to immediately return a value of **true**.

Other algorithms can find the answer quicker. This topic of searching a list is an area of active research in computer science.

```

/**
    Test if an integer is a prime.
 */
public boolean isPrime(int n)
{
    boolean answer = false;
    for (int prime : primes)      // for each prime
in primes
        if (n == prime)  // test prime
        {
            return true;
        }
    return answer;
}

```

Programming rule of thumb: Each method has only one **return** statement which is at the end.

A common programming approach is for the last command to return a default value. Thus the programmer is certain the method always returns an acceptable value.

Many programmers follow a stricter demand. They require that each method have only one return statement, which then must be the last command in the method. Thus our **isPrime** method sets the return value and uses a **break** command to exit any loop.

```

/**
    Test if an integer is a prime.
*/
public boolean isPrime(int n)
{
    boolean answer = false;
    for (int prime : primes)      // foreach prime
in primes
        if (n == prime)  // test prime
        {
            answer = true;
            break; // interrupt loop
        }
    return answer;
}

```

The Java compiler naturally requires a return statement at the end of each method. Notice this method does not return a value when the integer is zero.

```

/**
* Return +1 for positive integers
* Return -1 for negative integers
* Print "zero" for integer zero.
*/
public class PositiveInteger
{

    private int positiveInteger(int i)
    {
        if (i > 0) {

```

```

        return 1;
    }
    else if (i < 0) {
        return -1;
    }
    else {
        System.out.println("zero");
    }
}
}

```

The Java compiler produced this complaint.

```

----jGRASP exec: javac -g PositiveInteger.java

PositiveInteger.java:20: error: missing return
statement
    }
    ^
1 error

```

The Java compiler does not produce a complaint for this following method.

```

/**
 * Return +1 for positive integers
 * Return -1 for negative integers
 * Print 0 for integer zero.
 */
public class PositiveInteger
{

```

```

private int positiveInteger(int i)
{
    if (i > 0) {
        return 1;
    }
    else if (i < 0) {
        return -1;
    }
    else {
        return 0;
    }
}
}

```

Thus we can begin to understand the wisdom of the programmer's rule of thumb that each method has only one return statement and that the return statement is at the end of the method. So we would rewrite the method as follows. Because of the return statement at the end of the method, outside of any **if** or loop construct, the program certainly always returns a value.

```

/**
 * Return +1 for positive integers
 * Return -1 for negative integers
 * Print 0 for integer zero.
 */
public class PositiveInteger
{

```

```

private int positiveInteger(int i)
{
    int answer = 0;
    if (i > 0) {
        answer = 1;
    }
    else if (i < 0) {
        answer = -1;
    }
    else {
        answer = 0;
    }
    return answer;
}
}

```

Topic: Big O

Method **isPrime** uses a for-each loop to search the entire array. This method has a “Big O” classification (http://en.wikipedia.org/wiki/Big_O_notation) of $O(n)$ meaning as the array size of n grows the number of computer steps to locate the target value also grows like n . A computer algorithm of $O(n)$ is amongst the slowest of all algorithms and should be avoided, if possible.

```

/**
    Test if an integer is a prime.
 */
public boolean isPrime(int n)
{

```

```

    boolean answer = false;
    for (int prime : primes)        // foreach prime
in primes
        if (n == prime) // test prime
        {
            answer = true;
            break; // interrupt loop
        }
    return answer;
}

```

When the array is in sequence the method can use a binary search

(http://en.wikipedia.org/wiki/Binary_search_algorithm)

. A binary search begins by comparing the target value with the middle item in the array. If the target value is lower then the target is in the lower half of the array, otherwise the target is in the higher half. The search then focuses on the identified half. Thus a binary search repeatedly divides the range of the search in half until the target is located. The number of comparisons required is log to the base 2 of n. Thus the Big O notation for a binary search is $O(\log n)$. Searching an array of 100 items would require on average 50 comparisons using a sequential search. Two raised to the tenth power is 1,024 so a binary search requires at most only 10 comparisons. Computer programming is a form of mathematical reasoning applied to computing machinery. As you write your computer programs you should ponder their computing requirements.

This class exercises class Primes.

```
import java.util.Scanner;

/**
 * Test class Primes
 */
public class TestPrimeNumbers
{

    Primes p;

    /**
     * constructor
     */
    public TestPrimeNumbers()
    {
        p = new PrimeNumbers();
    }

    /**
     * This method lists all the prime numbers in the
    list.
     */
    private void list()
    {
        System.out.println("prime numbers in list:");
        int maxIndex = p.getMaxCntPrimes();
        for (int i = 0; i < maxIndex; i++)
        {
            System.out.println(p.getPrime(i)); // print
prime
        }
    }
}
```

```
/**
    This method prints the largest prime numbers in
the list.
*/
private void maxPrime()
{
    System.out.println("The largest prime in this
list: " + p.getMaxPrime());
}

/**
    This method prints the count of prime numbers in
the list.
*/
private void cntPrime()
{
    System.out.println("The count of prime numbers
in this list: " + p.getMaxCntPrimes());
}

/**
    This method checks whether or not an integer is
in the list of prime numbers.
*/
private void chkPrime()
{
    // Check 4 as a prime number
    System.out.print("prime candidate: ");
    int candidate = 4;
    System.out.print(candidate + " is ");
    if (p.isPrime(candidate) == false)
    {
```

```

        System.out.print("not ");
    }
    System.out.println("a prime number.");
    // Check 5 as a prime number
    System.out.print("prime candidate: ");
    candidate = 5;
    System.out.print(candidate + " is ");
    if (p.isPrime(candidate) == false)
    {
        System.out.print("not ");
    }
    System.out.println("a prime number.");
}

/**
    This method works with the prime numbers.
 */
private void go()
{
    Scanner input = new Scanner(System.in);
    final int EXIT = 0;
    final int LIST = 1;
    final int MAX = 2;
    final int CNT = 3;
    final int CHK = 4;
    int choice = -1;
    System.out.print("Testing class Primes \nSelect
1-List 2-Max 3-Count 4-prime 0-stop: ");
    choice = input.nextInt();
    while (choice > 0)
    {
        switch(choice) {
            case EXIT:

```

```

        System.exit(0);
        break;
    case LIST:
        list();
        break;
    case MAX:
        maxPrime();
        break;
    case CNT:
        cntPrime();
        break;
    case CHK:
        chkPrime();
        break;
    default:
        System.out.println("Enter 0, 1, 2, or
3.");
        break;
    }
    System.out.print("Testing class Primes
\nSelect 1-List 2-Max 3-Count 4-Prime 0-stop: ");
    choice = input.nextInt();
}
}

/**
    main method
    obj is a common abbreviation of "object".
*/
public static void main(String[] args)
{
    TestPrimes obj = new TestPrimes();
    obj.go();
}

```

```

    }

}

```

Topic: Scanner

Our discussion of new material begins with the **go** method. **Scanner input = new Scanner(System.in);** produces a new object of the **Scanner** class that is connected to **System.in**. Recall **System.out** is an output stream connected to our console. Likewise **System.in** is an input stream connected to our console. When you press the **enter** key the choice you typed on the console goes to the input stream named **System.in**. The **Scanner** object named **input** will parse that console string into its tokens which are separated by blank spaces. A space is the default delimiter separating tokens. The next command, **choice = input.nextInt();**, tells that **Scanner** object to search the tokens from the input string for the next integer. The switch statement then branches depending on the integer of your choice.

Topic: Scope

Notice how the **Scanner** class object named **input** is defined in the **go** method. The **Scanner** class object named **input** is used only in the **go** method, so its defined in the **go** method. Thus only the java commands within method **go** can use the **Scanner** class object named **input**. The scope of **input** is the method named **go**. When a variable or object can only be seen, thus

used, only in one method, then that variable or object has a scope of that method.

Programmers define their variables and objects within the smallest possible scope. Restricting the scope of variables prevent accidental misuse of that variable.

Topic: Magic Values

Instead of “magic numbers” such as the integers 0, 1, 2, and 3 programmers use constants. More discussion on this topic is available on Wikipedia at [http://en.wikipedia.org/wiki/Magic_number_\(programming\)](http://en.wikipedia.org/wiki/Magic_number_(programming)) under the title of “Unnamed numerical constants”. Thus in the program listed above the switch statement has cases using constants, not integers. Those constants are defined using final, which means the program can only assign a value to that variable once. From then on that value cannot be changed.

```
final int EXIT = 0;
final int LIST = 1;
final int MAX = 2;
final int CNT = 3;
final int CHK = 4;
```

Using named constants is preferable to using integer values as shown below.

```
/**
    This method works with the prime numbers.
*/
```

```
private void go()
{
    Scanner input = new Scanner(System.in);
    System.out.print("Testing class Primes \nSelect
1-List 2-Max 3-Count 4-Prime 0-stop: ");
    choice = input.nextInt();
    while (choice > 0)
    {
        switch(choice) {
            case 0:
                System.exit(0);
                break;
            case 1:
                list();
                break;
            case 2:
                maxPrime();
                break;
            case 3:
                cntPrime();
                break;
            case 4:
                chkPrime();
                break;
            default:
                System.out.println("Enter 0, 1, 2, 3 or
4.");
                break;
        }
        System.out.print("Testing class Primes
\nSelect 1-List 2-Max 3-Count 4-Prime 0-stop: ");
        choice = input.nextInt();
    }
}
```

```
}
```

Using constants instead of magic numbers makes your program easier to understand. Most of the work of computer programming is updating and maintaining computer programs. Seldom does a computer programmer develop an entirely new computer program. Using case statements for 0, 1, 2, 3, and 4 is easier to type, but using named constants saves effort in the long term.

The **default** case can appear anywhere in the list of cases, but is most commonly used as the last case.

The method **maxPrime** displays the sentence: "The largest prime in this list is 161". To produce that line **maxPrime** called the **getMaxPrime** method of the **PrimeNumbers** object named **p**.

```
/**
    This method prints the largest prime numbers in
    the list.
 */
private void maxPrime()
{
    System.out.println("The largest prime in this
list: " + p.getMaxPrime());
}
```

Because method **maxPrime** has only one command that method could have been eliminated by simply moving the one command to the **switch** statement.

```
switch(choice) {
```

```
    case EXIT:
        System.exit(0);
        break;
    case LIST:
        list();
        break;
    case MAX:
        // maxPrime();
        System.out.println("The largest prime in
this list: " + p.getMaxPrime());
        break;
    case CNT:
        cntPrime();
        break;
    case CHK:
        chkPrime();
        break;
    default:
        System.out.println("Enter 0, 1, 2, or
3.");
        break;
}
```

Most programmers prefer to keep the case statements as brief as possible. Also using a method call makes the program easier to read and understand. Using a separate method instead of writing the print command in the case statement is a question of judgment. Certainly programmers avoid large amounts of code in a case statement.

Topic: When to use the for, while, and for-each

commands.

The `list` method displays all the prime numbers in the array. This method gets the size of the array from the `getMaxCntPrimes` method of the `PrimeNumbers` object. Because the size of the array is known, this method uses a **for** loop as shown below.

```

    /**
        This method lists all the prime numbers in the
    list.
    */
    private void list()
    {
        System.out.println("prime numbers in list:");
        int maxIndex = p.getMaxCntPrimes();
        for (int i = 0; i < maxIndex; i++)
        {
            System.out.println(p.getPrime(i)); // print
prime
        }
    }

```

The `go` method uses a while loop because the number of iterations needed is unknown. The while loop continues iterating the loop until the sentinel value is met. In this case the **while** loop continues until the user selects a choice of 0 to exit the program.

Another method named `getArrayPrimes` could return the address of the array of prime numbers.

```

    /**

```

This method the entire array of the prime numbers.

```
*/  
public int[] getArrayPrimes()  
{  
    return primes;  
}
```

The corresponding method in the test class calls the **getArrayPrimes** method of the **PrimeNumbers** object. Then the list method can use a for-each command to loop through the contents of that array. The source code is shown here.

```
/**  
    This method lists all the prime numbers in the  
list.  
*/  
private void list()  
{  
    System.out.println("prime numbers in list:");  
    int[] primes = p.getArrayPrimes();  
    for (int prime : primes)  
    {  
        System.out.println(prime); // print prime  
    }  
}
```

Overall these two classes have shown examples of the for, while, and for-each commands.

Topic: pass by reference versus pass by value

When a primitive value, such as an `int`, is returned a copy of the value is returned. Thus the calling object gets the value, but cannot change the value stored in the called object. For example the original `list` method received each prime number as an `int` value.

Example: pass by value

Then **TestPrime** could erroneously include a command to attempt to reset each prime number to zero.

```

/**
    This method lists all the prime numbers in the
list.
    This method contains an error.
*/
private void list()
{
    System.out.println("prime numbers in list:");
    int maxIndex = p.getMaxCntPrimes();
    for (int i = 0; i < maxIndex; i++)
    {
        int prime = p.getPrime(i);
        System.out.println(prime); // print prime
        /* The following command changes the value of
the local          variable named prime. It does
not change the value in the          PrimeNumbers
object.

        */
        prime = 0;
    }
}

```

The command, `prime = 0;`, attempts to reset the prime number in object `PrimeNumbers`, but only changes the local variable named `prime`. The prime number in the `PrimeNumbers` object named `p` is not changed. When returning a primitive value, such as an `int`, the method `getPrime` of object `PrimeNumbers` returns a copy of the value, so the calling object named `TestPrimeNumbers` can only change its local variable, not the original prime number in object `PrimeNumbers`.

Example: pass by reference

The `getArrayPrimes` method of the `PrimeNumbers` object returns a reference, also known as a pointer, to the actual array of prime numbers held within the `PrimeNumbers` object. Then the following method can and will change all the prime numbers to zero.

```

/**
    This method lists all the prime numbers in the
list.
*/
private void list()
{
    System.out.println("prime numbers in list:");
    int[] primes = p.getArrayPrimes();
    for (int i = 0; i < primes.length; i++)
    {
        int prime = primes[i];
        System.out.println(prime); // print prime
        // Change the prime to zero.
        primes[i] = 0;
    }
}

```

```
}
```

The `primes[i] = 0;` command within class `TestPrimeNumbers` directly changes the value of the array element within the other object named `p` of class type `PrimeNumbers`. The command `int[] primes = p.getArrayPrimes();` meant `primes` got a reference to the array of prime numbers in object `p`, which was of class type `PrimeNumbers`, so the command `primes[i] = 0;` could actually change values of the prime numbers stored in the `PrimeNumbers` object named `p`.

This method produced these results during testing. First the prime numbers are listed. The second request for a list produced an equally long list of zeroes.

Testing class Primes

```
Select 1-List 2-Max 3-Count 4-prime 0-stop: 1
```

```
prime numbers in list:
```

```
1
```

```
2
```

```
3
```

```
...
```

```
181
```

Testing class Primes

```
Select 1-List 2-Max 3-Count 4-Prime 0-stop: 1
```

```
prime numbers in list:
```

```
0
```

```
0
```

```
0
```

```
...
```

```
0
```

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop:

Clearly pass by value produces different results from pass by reference. Thus programmers carefully limit their use of pass by reference. For that reason the **getArrayPrimes** method was not included in the original version of class **PrimeNumbers**.

The power of object-oriented programming comes from using objects. Methods do normally return objects, thus methods do normally use pass by reference.

Topic: Test the limits

Currently the test class only checks if the numbers 4 and 5 are prime numbers. The method's commands have been rewritten as follows.

```
/**
    This method checks whether or not an integer is
    in the list of prime numbers.
 */
private void chkPrime()
{
    System.out.print("prime candidate: ");
    int candidate = input.nextInt();
    System.out.print(candidate + " is ");
    if (p.isPrime(candidate) == false)
    {
        System.out.print("not ");
    }
}
```

```

        System.out.println("a prime number.");
    }

```

The command `int candidate = input.nextInt();` gets the next int value from the scanner object named `input`, which is linked to the input stream `System.in`. The JVM displays the following compiler error message.

```
TestPrimeNumbers.java:73: error: cannot find symbol
```

```
int candidate = input.nextInt();
```

```
symbol:    variable input
```

```
location: class TestPrimeNumbers
```

```
1 error
```

The scanner object named `input` is defined within the `go` method, so its scope is limited to that method. Moving the declaration of scanner `input` to the top of the class resolves the compilation error.

```
import java.util.Scanner;
```

```
/**
```

```
 * Test class Primes
```

```
 */
```

```
public class TestPrimeNumbers
```

```
{
```

```
    PrimeNumbers p;
```

```
    Scanner input;
```

```
    /**
```

```
*   constructor
*/
public TestPrimeNumbers()
{
    p = new PrimeNumbers();
    input = new Scanner(System.in);
}
```

Also the constructor now initializes scanner **input** to a new scanner object linked to the input stream named **System.in**. Because the scanner object named **input** now has class-wide scope, the class compiles correctly.

A large amount of effort is required to ensure your computer program's answers are correct. The field of testing is a major research area in its own right.

Beginning programmers normally exercise their computer programs for a few sample values. When those samples agree with your predicted values then they declare their computer programs to be correct. While checking your computer program you should test the limits. For this problem the limits are the smallest prime number 1 and the largest prime number in the list 181. Notice that the console's output for this computer program includes both the limits and a few sample numbers when checking for prime numbers.

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop: 4

prime candidate: 1

1 is a prime number.

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop: 4

prime candidate: 181

181 is a prime number.

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop: 4

prime candidate: 4

4 is not a prime number.

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop: 4

prime candidate: 23

23 is a prime number.

Testing class Primes

Select 1-List 2-Max 3-Count 4-Prime 0-stop:

Hint: Text-based test programs can lead to rapid debugging

Since text-based programs can be rapidly developed they are an excellent method for quickly checking and debugging your computer programs.

Topic: Recursion

Recursion means a method repeatedly calls itself. The binary search algorithm lends itself to recursion. The algorithm begins searching the entire range from the first to the last by checking the middle location. If the candidate equals that middle prime number then the search is finished. Next the algorithm decides whether to search the lower or higher section of the list. So the algorithm repeated performs the same task simply on smaller portions of the list of prime numbers.

A recursive algorithm normally begins by testing for the end state. This algorithm has two end states. When the candidate equals the middle item in the list then the algorithm returns a boolean completion code of true. Thus the top line calculates the location of the middle item. Next the comparison and return statements.

```
int middle = low + ((high - low) / 2);
if (candidate == primes[middle])
{
    return true;
}
```

Next the algorithm checks for the state when the candidate is not a prime. The algorithm continues checking smaller and smaller intervals designated by the two variable low and high. Normally during each recursive call low approaches high. If low is greater than or equal to high, then the candidate never matched the middle item, so it's not in the list.

```
if (low >= high)
{
    return false;
}
```

This program has just completed two different tests for completion. Notice that the previous two tests must be completed in the sequence described.

Next the algorithm must decide whether to search the lower half or the higher half of the range. Comparing

the candidate with the middle item gives the answer. If the candidate is less than the middle item, then the candidate must be in the lower range.

```
        if (candidate < primes[middle])
        {
            return binarySearchPrimes(low, middle-1,
candidate);
        }
```

The next range is from the low to middle - 1, not low to middle because the algorithm has already checked the middle item in the list.

Likewise checking the upper portion of the range would be from the middle + 1 item to the top item.

```
        return binarySearchPrimes(middle+1, high,
candidate);
```

Because of the conditional statements the algorithm is written as a series of nested **if** statements.

```
    /**
     * Use a binary search to find whether or not an
integer is a prime number.
     */
    private boolean binarySearchPrimes(int low, int
high, int candidate)
    {
        int middle = low + ((high - low) / 2);
        if (candidate == primes[middle])
        {
```

```

        return true;
    }
    else
    {
        if (low >= high)
        {
            return false;
        }
        else
        {
            if (candidate < primes[middle])
            {
                return binarySearchPrimes(low, middle-
1, candidate);
            }
            else
            {
                return binarySearchPrimes(middle+1,
high, candidate);
            }
        }
    }
}

```

Because the nested **if** statements march off the page to the right, thus making the program difficult to read and comprehend, many programmers combine the **else** and **if** commands on one line. Then the nested **if** statements march down the page, not across. This style of coding only works when the program uses only one of the two result branches for each **if** statement.

/**

```

    * Use a binary search to find whether or not an
    integer is a prime number.
    */

    private boolean binarySearchPrimes(int low, int
high, int candidate)
    {
        boolean rtnValue = true;
        int middle = low + ((high - low) / 2);
        // Check for completion when match is found
        if (candidate == primes[middle])
        {
            rtnValue = true;
        }
        // Check for completion when candidate is not in
list
        // This test must be after, not before, the
previous test
        else if (low >= high)
        {
            rtnValue = false;
        }
        else if (candidate < primes[middle])
        {
            // Recursive call using lower half of list
            rtnValue = binarySearchPrimes(low, middle-1,
candidate);
        }
        else
        {
            // Recursive call using higher half of list
            rtnValue = binarySearchPrimes(middle+1, high,
candidate);
        }
    }

```

```

    return rtnValue;
}

```

The block symbols { and } are only needed when more than one statement is in the then or else group. Thus the code can be written much more briefly as follows.

```

/**
 * Use a binary search to find whether or not an
 * integer is a prime number.
 */
private boolean binarySearchPrimes(int low, int
high, int candidate)
{
    boolean rtnValue = true;
    int middle = low + ((high - low) / 2);
    if (candidate == primes[middle])
        rtnValue = true;
    else if (low >= high)
        rtnValue = false;
    else if (candidate < primes[middle])
        rtnValue = binarySearchPrimes(low, middle-1,
candidate);
    else
        rtnValue = binarySearchPrimes(middle+1, high,
candidate);
    return rtnValue;
}

```

Surrounding the then and else statements in block symbols { } makes your code safer. Adding a statement to either a then or else group of statements would require adding the block symbols { and }. Forgetting

those { and } symbols would result in a compilation error. Thus the norm for an **if** statement is shown below.

```
if (comparison)
    statement;
else
    statement;
```

Most programmers and IDEs routinely add the block symbols { and }.

```
if (condition)
{
    statement;
}
else
{
    statement;
}
```

Many programmers find recursive programs easier to write and understand. The model is two steps. First test for the state of completion. If the computation is not complete then update the parameters for the next call of the recursive program.

To better understand the stack of execution calls to the recursive method programmers will include a leading series of dots before each print message. As the recursions increase in depth of calls the string of dots grows larger.

Topic: Include print statements to display program execution.

Notice these print statements.

```
        System.out.print("candidate: " + candidate + "  
low: " + low + " high: " + high);  
        System.out.println(" middle: " + middle);
```

They are not required by the algorithm. They help the programmer find any errors in the method's implementation of the algorithm. Each time the method is called the print statements display the values in important variable such as the list of parameters. Programmers either depend on the IDE's debugging tools, or they include print statements. Either method helps the programmer trace the execution of the method's statements. Any discrepancy between the printed low, high, and middle values and the expected values indicates an error.

To help us better comprehend recursion the program prints the low, high, and middle values. Also an offset, which is two dots, is printed at the start of each recursive call. As the recursion continues the offsets grows by an additional two dots. The recursive method has been modified to also display the return values.

```
/**  
 * Use a binary search to find whether or not an
```

```

integer is a prime number.
    */
    private boolean binarySearchPrimes(String offset,
int low, int high, int candidate)
    {
        int middle = low + ((high - low) / 2);
        System.out.print(offset + "candidate: " +
candidate + " low: " + low + " high: " + high);
        System.out.println(" middle: " + middle);
        // Check for completion when match is found
        if (candidate == primes[middle])
        {
            System.out.println(offset + "return true");
            return true;
        }
        else
        {
            // Check for completion when candidate is not in
list
            // This test must be after, not before, the
previous test
            if (low >= high)
            {
                System.out.println(offset + "return
false");
                return false;
            }
            else
            {
                offset += "..";
                if (candidate < primes[middle])
                {
                    // Recursive call using lower half of list

```

```

        boolean rtnValue =
binarySearchPrimes(offset, low, middle-1, candidate);
        System.out.println(offset + rtnValue);
        return rtnValue;
    }
    else
    {
        // Recursive call using higher half of list
        boolean rtnValue =
binarySearchPrimes(offset, middle+1, high, candidate);
        System.out.println(offset + rtnValue);
        return rtnValue;
    }
}
}
}

```

At the top of the method the print statements display the list of parameters: offset, low, and high. Also the location of the middle item is shown.

```

        System.out.print(offset + "candidate: " +
candidate + " low: " + low + " high: " + high);
        System.out.println(" middle: " + middle);

```

When the candidate matches the middle item a print statement displays "return true".

```

    if (candidate == primes[middle])
    {
        System.out.println(offset + "return true");
        return true;
    }

```

Before recursively calling itself the method increase the size of offset by an additional two dots.

```

        offset += "..";
        if (candidate < primes[middle])
        {
            boolean rtnValue = binarySearchPrimes(offset,
low, middle-1, candidate);
            System.out.println(offset + rtnValue);
            return rtnValue;
        }

```

The revised method produces this report.

Testing class Primes

Select 1-List 2-Max 3-Count 4-prime 0-stop: 4

prime candidate: 5

5 is

..candidate: 5 low: 0 high: 60 middle: 30

....candidate: 5 low: 0 high: 29 middle: 14

.....candidate: 5 low: 0 high: 13 middle: 6

.....candidate: 5 low: 0 high: 5 middle: 2

.....candidate: 5 low: 3 high: 5 middle: 4

.....candidate: 5 low: 3 high: 3 middle: 3

.....return true

.....true

.....true

.....true

.....true

....true

a prime number.

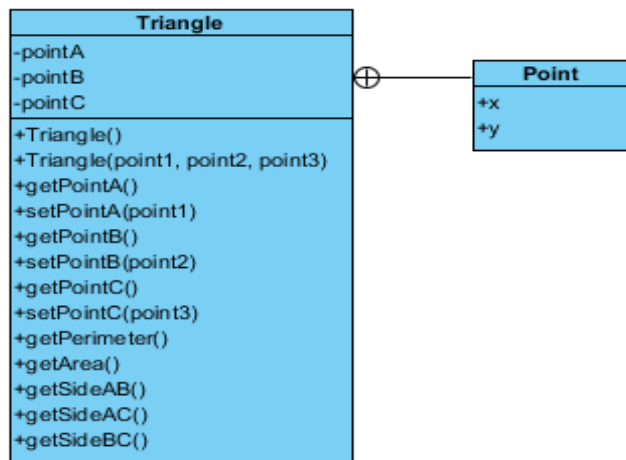
Chapter Seven: java interface

Topic: UML

Unified Modeling Language (UML), described in Wikipedia at

http://en.wikipedia.org/wiki/Unified_Modeling_Language

, is a standard for describing class diagrams. This chapter's project will work with multiple classes, so a UML diagram would help us understand the interactions between those classes.



UML class diagrams display the class name at the top. A horizontal line separates that class name from a list of attributes. An additional horizontal line then separates the methods from the attribute list. So class diagrams have three main sections: name, attributes, and methods.

Before each attribute and method is a indicator of the access limitations. A dash (-) indicates private access. Only methods within that same class can access private items, such as attributes and other methods. A

plus symbol (+) stands for public access. Other objects and their methods can use items with public access. And, a # is used for protected access, which limits access to objects and their methods in the same package. A java package corresponds to a computer's folder.

The Triangle class uses three points to describe each triangle. Since those three points are attributes, the access is set to private. Both a default constructor and fully parametrized constructors are available. The default constructor creates three points all at location (0,0). The fully parametrized constructor simply uses the three points available in the list of arguments.

The Point class uses the java standard **Point2D.Double** class using double values for x and y. A different Point class uses int values for x and y. Point2D is an abstract class. The java Application Program Interfaces (APIs) describe this hierarchy of classes at <http://docs.oracle.com/javase/7/docs/api/>.

Each Triangle object contains three objects of class Point so the connecting line from class diagram for Triangle to class diagram for Point begins with a special symbol. Other authors may use different symbols to represent the connecting relationship in their UML diagrams. The UML diagram shown above was drawn using the community edition of Visual Paradigm described at <http://www.visual-paradigm.com/>. Much more detail can be included in UML class diagrams. The UML diagram shown above is for introducing use of UML.

The standard strategy for developing a computer application begins with drawing UML class diagrams for the core classes, but not the test classes. Utility classes such as Math and Scanner are also not included. Commercial versions of UML can automatically produce java source code for those classes, which greatly reduces the programmer's effort.

The next stage of development writes and tests java programs to the core classes. Here is the source code for the Triangle class.

```
import java.awt.geom.*;

/**
 * Class Triange uses three Points to describe its
 * triangle.
 */

public class Triangle
{

    // Attributes
    Point2D.Double pointA, pointB, pointC;

    // Constructors

    /**
     * default constructor
     */
    public Triangle()
    {
```

```
        pointA = new Point2D.Double(); // by default set
to (0,0)
        pointB = new Point2D.Double();
        pointC = new Point2D.Double();
    }

    /**
     * fully parametrized constructor
     */
    public Triangle(Point2D.Double a, Point2D.Double b,
Point2D.Double c)
    {
        pointA = a;
        pointB = b;
        pointC = c;
    }

    // Accessors and mutators

    /**
     * getPointA returns the Point object for attribute
pointA
     */
    public Point2D.Double getPointA()
    {
        return pointA;
    }

    /**
     * setPointA sets pointA to the Point provided in
the list of arguments */
    public void setPointA(Point2D.Double point)
    {
```

```
        pointA = point;
    }

    /**
     * getPointB returns the Point object for attribute
pointB
     */
    public Point2D.Double getPointB()
    {
        return pointB;
    }

    /**
     * getPointB sets pointB to the Point provided in
the list of arguments */
    public void setPointB(Point2D.Double point)
    {
        pointB = point;
    }

    /**
     * getPointC returns the Point object for attribute
pointC
     */
    public Point2D.Double getPointC()
    {
        return pointC;
    }

    /**
     * getPointA sets pointA to the Point provided in
the list of arguments
     */
```

```
public void setPointC(Point2D.Double point)
{
    pointC = point;
}

/**
 * get length of side A - B
 */
public double getSideAB()
{
    // Point2D.Double extends Point2D which provides
the distance method
    return pointA.distance(pointB);
}

/**
 * get length of side A - B
 */
public double getSideBC()
{
    return pointB.distance(pointC);
}

/**
 * get length of side A - B
 */
public double getSideAC()
{
    return pointA.distance(pointC);
}

/**
 * get perimeter of triangle
```

```

*/
public double getPerimeter()
{
    return getSideAB() + getSideBC() + getSideAC();
}

/**
 * calculate area of triangle
 * In geometry, Heron's (or Hero's) formula, named
after Heron of Alexandria,
 * states that the area T of a triangle whose sides
have lengths a, b, and c is
 *  $area = \sqrt{s(s-a)(s-b)(s-c)}$ 
 * where s is the semiperimeter of the triangle.
 * Source: http://en.wikipedia.org/wiki/Hero
%27s\_formula
 */
public double getArea()
{
    double s = getPerimeter() / 2;
    double a = getSideAB();
    double b = getSideBC();
    double c = getSideAC();
    double area = Math.sqrt(s * (s-a) * (s-b) * (s-
c));
    return area;
}
}

```

This command `import java.awt.geom.*;` allows the compiler to freely import classes from the geom directory of the Abstract Windows Toolkit (AWT)

directory. The asterisk * means all the classes within the java.awt.geom directory are available for inclusion as needed. Classes are only included when necessary. Thus programmers often use the asterisk, since only the needed classes are included. This differs from other computer languages.

The Triangle class has three attributes all of class Point2D.Double, thus the following declaration is allowed.

```
// Attributes  
Point2D.Double pointA, pointB, pointC;
```

That one declaration is equivalent to the following three declarations.

```
// Attributes  
Point2D.Double pointA;  
Point2D.Double pointB;  
Point2D.Double pointC;
```

The class name Point2D.Double includes a dot (.) because class Double is defined within class Point2D as can be seen in the java docs for that class.

Overview	Package	Class	Use	Tree	Deprecated	Index	Help
Prev Class	Next Class	Frames	No Frames	All Classes			
Summary: Nested Field Constr Method				Detail: Field Constr Meth			

java.awt.geom

Class Point2D.Double

java.lang.Object

java.awt.geom.Point2D

java.awt.geom.Point2D.Double

All Implemented Interfaces:

Serializable, Cloneable

Enclosing class:

Point2D

```
public static class Point2D.Double
    extends Point2D
    implements Serializable
```

The `Double` class defines a point specified in `double` precision.

Notice the API's statement that the enclosing class is `Point2D`, thus class `Double` is defined within class `Point2D`. The java code would appear as shown below.

```
public class Point2D
{
    ...
    public class Double
    {
        ...
    }
}
```

```
...
}
```

Next in the java class Triangle are the constructors. The default constructor creates three separate points and assigns them to the variables point A, point B, and point C.

```
/**
 * default constructor
 */
public Triangle()
{
    pointA = new Point2D.Double(); // by default set
to (0,0)
    pointB = new Point2D.Double();
    pointC = new Point2D.Double();
}
```

The default constructor for the Double class, which is defined within class Point2D, automatically creates a point at location (0,0). Although all three points are at (0,0), the constructor must create three different point objects. The constructor cannot simply set all three variables pointA, pointB, and pointC to a single point.

```
/**
 * default constructor
 * This constructor contains a serious ERROR
 */
public Triangle()
{
```

```
    pointA = new Point2D.Double();
    pointB = pointA;
    pointC = pointA;
}
```

The fully parametrized constructor must individually define each argument in its list of arguments, even though all are of the same type: `Point2D.Double`.

```
/**
 * fully parametrized constructor
 */
public Triangle(Point2D.Double a, Point2D.Double b,
Point2D.Double c)
{
    pointA = a;
    pointB = b;
    pointC = c;
}
```

The rules of Java do not permit any abbreviation of the list of arguments such as shown below.

```
/**
 * fully parametrized constructor has an ERROR in
 * its list of arguments
 * Each argument requires its own type declaration.
 */
public Triangle(Point2D.Double a, b, c)
{
    pointA = a;
    pointB = b;
    pointC = c;
}
```

```
}
```

The access methods for the lengths of the sides of the triangle all depend on the distance method provided by the **Point2D** class. Each point is of subclass **Point2D.Double** so they inherit the **distance** method from its parent class **Point2D**.

```
/**
 * get length of side A - B
 */
public double getSideAB()
{
    // Point2D.Double extends Point2D which provides
the distance method
    return pointA.distance(pointB);
}
```

The **getArea** method implements Hero's formula. Many other equations are available for calculating the area of a triangle.

```
/**
 * calculate area of triangle
 * In geometry, Heron's (or Hero's) formula, named
after Heron of Alexandria,
 * states that the area T of a triangle whose sides
have lengths a, b, and c is
 *  $T = \sqrt{s(s-a)(s-b)(s-c)}$ 
 * where s is the semiperimeter of the triangle.
 * Source: http://en.wikipedia.org/wiki/Hero's\_formula
 */
```

```
public double getArea()
{
    double s = getPerimeter() / 2;
    double a = getSideAB();
    double b = getSideBC();
    double c = getSideAC();
    double area = Math.sqrt(s * (s-a) * (s-b) * (s-
c));
    return area;
}

}
```

The test program uses a text-based console program because such programs are quick and easy to generate.

```
import java.util.Scanner;
import java.awt.geom.*;

/**
 * Test calculator of triangle facts
 */

public class TestTriangle
{

    /**
     * Default constructor
     */
    public TestTriangle()
    {

    }

}
```

```

/**
 * Go method
 */
private void go()
{
    System.out.println("Start test of Triangle
object.");

    Point2D.Double pointA = new Point2D.Double(0,0);
    Point2D.Double pointB = new Point2D.Double(0,1);
    Point2D.Double pointC = new Point2D.Double(1,1);
    Triangle tri = new Triangle(pointA, pointB,
pointC);

    Point2D.Double a = tri.getPointA();
    Point2D.Double b = tri.getPointB();
    Point2D.Double c = tri.getPointC();

    System.out.println("Point A: (" + a.getX() + ","
+ a.getY() + ")");

    System.out.println("Point B: (" + b.getX() + ","
+ b.getY() + ")");

    System.out.println("Point C: (" + c.getX() + ","
+ c.getY() + ")");

    System.out.println("side AB: " + tri.getSideAB()
);

    System.out.println("side BC: " + tri.getSideBC()
);

    System.out.println("side AC: " + tri.getSideAC()
);

    System.out.println("perimeter: " +
tri.getPerimeter());

    System.out.println("area: " + tri.getArea());
}

```

```

/**
 * Main method
 */
public static void main(String[] args)
{
    System.out.println("Start test run of
calculator.");
    TestTriangle obj = new TestTriangle();
    obj.go();
}
}

```

The test program produces the following output.

```

Start test run of calculator.
Start test of Triangle object.
Point A: (0.0,0.0)
Point B: (0.0,1.0)
Point C: (1.0,1.0)
side AB: 1.0
side BC: 1.0
side AC: 1.4142135623730951
perimeter: 3.414213562373095
area: 0.49999999999999983

```

¶

The test triangle is a right triangle with sides of 1, 1, and square root of 2, so the results for side AB and side BC are correct. Side AC should be the value for the square root of two, which Wikipedia quotes OEIS as

1.41421356237309504880168872420969807856967187537694807317667973799... (sequence A002193 in OEIS), so the

program's answer is correct to the number of decimal places displayed. The area should be one-half, since the area is one-half of a one by one square, thus the answer for the area is wrong beginning with the seventeenth and eighteenth decimal positions, which contain 83. All the float and decimal operations contain small errors, which accumulate as the calculations are performed. Let's focus on the trail of square root operations. **Triangle** class called the **Math** class to calculate the square root of $s(s-a)(s-b)(s-c)$. Variables *a*, *b*, and *c* had been calculated by calling the **distance** method of the **Point2D.Double** class, which might have called the square root method, so **getArea's** calculation might have involved three more square root operations to calculate *a*, *b*, and *c* for a total of four square root operations. Instead of having each slight error balance out another, using the same square root operation could have led to a steady increase of the total error.

Topic: printf

Many programmer use the **printf** command to display numeric values to a reasonable amount of accuracy. The **printf** command appears in many other computer programming languages, with java being one of the languages most recently adding support for **printf**. Wikipedia provides an overview at <http://en.wikipedia.org/wiki/Printf>.

The test triangle class now includes the following command.

```
System.out.printf("printf area: %7.5f",  
tri.getArea() );
```

The above command begins with a string enclosed in quotation marks: `"printf area: %7.5f"`, which displays each character on the print line until an `%` symbol is encountered. The `%` symbol provides special processing commands. `"7.5"` means the output will be seven characters wide including a decimal point and five decimal places. The `"f"` means a non-integer number, such as java's `double` or `float` numbers. Of course numbers are rounded to the specified decimal place.

The test program now produces these results.

Start test run of calculator.

Start test of Triangle object.

Point A: (0.0,0.0)

Point B: (0.0,1.0)

Point C: (1.0,1.0)

side AB: 1.0

side BC: 1.0

side AC: 1.4142135623730951

perimeter: 3.414213562373095

area: 0.49999999999999983

printf area: 0.50000

The Open Group at

<http://www.opengroup.org/susv3xsh/fprintf.html>

provides a detailed description of the options available for displaying data using the **printf** method.

For the next iteration of the text-based test class includes a loop, so more than one triangle can be

tested. Also a Scanner class is used to get the user's x and y coordinates for each point of the triangles.

```
import java.util.Scanner;
import java.awt.geom.*;

/**
 * Test calculator of triangle facts
 */

public class TestTriangleV2
{
    // attributes
    Triangle tri;

    Scanner scan;

    /**
     * Default constructor
     */
    public TestTriangleV2()
    {
        // initialize scanner
        scan = new Scanner(System.in);
        // Create default triangle
        Point2D.Double pointA = new Point2D.Double(0,0);
        Point2D.Double pointB = new Point2D.Double(0,1);
        Point2D.Double pointC = new Point2D.Double(1,1);
        Triangle tri = new Triangle(pointA, pointB,
pointC);
    }

    /**
```

```
* makePoint sets up a new Point2D.Double
*/
private Point2D.Double makePoint()
{
    double x = 0;
    double y = 0;
    Point2D.Double point = new Point2D.Double(0,0);
    System.out.println("Creating new (x,y) point");
    System.out.print("Enter x:");
    x = scan.nextDouble();
    System.out.print("Enter y:");
    y = scan.nextDouble();
    point = new Point2D.Double(x,y);
    return point;
}

/**
 * makeTriangle sets up a new triangle.
 */
private void makeTriangle()
{
    Point2D.Double a = makePoint();
    Point2D.Double b = makePoint();
    Point2D.Double c = makePoint();
    tri = new Triangle(a,b,c);
}

/**
 * listTriPoints displays the (x,y) coordinates of
all three vertices
 */
private void listTriPoints()
{

```

```

    Point2D.Double a = tri.getPointA();
    Point2D.Double b = tri.getPointB();
    Point2D.Double c = tri.getPointC();
    System.out.println("Triangle");
    System.out.printf("Vertex\tx\ty\n");
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'A',
a.getX(), a.getY());
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'B',
b.getX(), b.getY());
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'C',
c.getX(), c.getY());
}

/**
 * Go method
 */
private void go()
{
    boolean done = false;
    System.out.println("Start test of Triangle
object.");
    System.out.print("Enter 0-exit 1-setup 2-list 3-
perimeter 4-area: ");
    int choice = scan.nextInt();
    while (choice != 0)
    {
        switch(choice)
        {
            case 0:
                System.out.println("bye");
                System.exit(0);
            case 1:
                makeTriangle();

```

```

        break;
    case 2:
        listTriPoints();
        break;
    case 3:
        System.out.printf("perimeter: %.3f\n",
tri.getPerimeter());
        break;
    case 4:
        System.out.printf("area: %.3f\n",
tri.getArea());
        break;
    default:
        System.out.println("Enter 0, 1, 2, 3,
or 4");
    }
    System.out.print("Enter 0-exit 1-setup 2-list
3-perimeter 4-area: ");
    choice = scan.nextInt();
}

}

/**
 * Main method
 */
public static void main(String[] args)
{
    System.out.println("Start test run of
calculator.");
    TestTriangleV2 obj = new TestTriangleV2();
    obj.go();
}

```

```
}

```

Now the program produced this report.

Start test run of calculator.

Start test of Triangle object.

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 1

Creating new (x,y) point

Enter x:0

Enter y:0.0

Creating new (x,y) point

Enter x:2.5

Enter y:3.1

Creating new (x,y) point

Enter x:7.25

Enter y:8.3725768

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 2

Triangle

Vertex	x	y
A	0.000	0.000
B	2.500	3.100
C	7.250	8.373

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 3

perimeter: 22.154

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 4

area: 0.772

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 0

The list points method contains printf commands to display a table. Of special note are the tab "\t" and new line "\n" commands. The backslash character, known as escape characters, allows special formatting

controls.

```

    /**
     * listTriPoints displays the (x,y) coordinates of
     all three vertices
     */
    private void listTriPoints()
    {
        Point2D.Double a = tri.getPointA();
        Point2D.Double b = tri.getPointB();
        Point2D.Double c = tri.getPointC();
        System.out.println("Triangle");
        System.out.printf("Vertex\tx\ty\n");
        System.out.printf("%c\t%5.3f\t%5.3f\n", 'A',
a.getX(), a.getY());
        System.out.printf("%c\t%5.3f\t%5.3f\n", 'B',
b.getX(), b.getY());
        System.out.printf("%c\t%5.3f\t%5.3f\n", 'C',
c.getX(), c.getY());
    }

```

Notice the list points method produced this table.

Triangle

Vertex	x	y
A	0.000	0.000
B	2.500	3.100
C	7.250	8.373

The command `printf("%c\t%5.3f\t%5.3f\n", 'A', a.getX(), a.getY());` produced the line "A 0.000 0.000". The `printf`'s list of arguments begins with a command string: `"%c\t%5.3f\t%5.3f\n"`. For each of the

following arguments, the command string includes a corresponding placeholder designated by a % symbol. The %c placeholder matches with the 'C' character. The \t escape character is a tab command. The two %f placeholders match the following two numbers provided by methods **a.getX()** and **a.getY()**. The %5.3f placeholder matches a java double or float number. The 5.3 specifies a total width of five columns with three digits to the right of the decimal point. The whole number portion is thus restricted to a single digit. Obviously that is not sufficient. The use of tabs contributes to the appearance of a table with a title line, column header line, and three rows of data. Each row describes a single vertex of the triangle.

Topic: Exception

Continued testing produced the error message displayed next.

Start test of Triangle object.

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 2.3

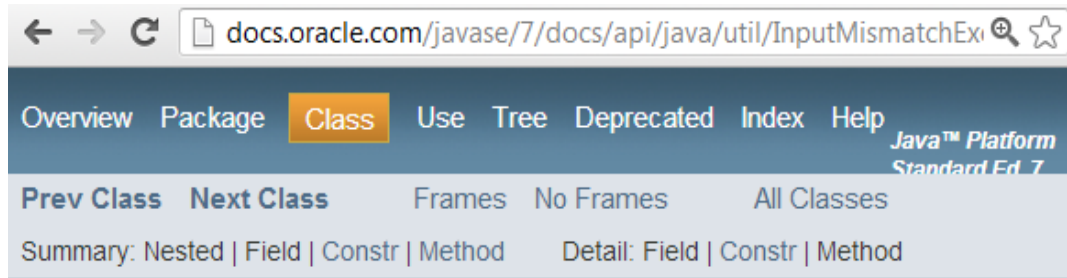
Exception in thread "main"

java.util.InputMismatchException

```
at java.util.Scanner.throwFor(Scanner.java:909)
at java.util.Scanner.next(Scanner.java:1530)
at java.util.Scanner.nextInt(Scanner.java:2160)
at java.util.Scanner.nextInt(Scanner.java:2119)
at TestTriangleV2.go(TestTriangleV2.java:80)
at TestTriangleV2.main(TestTriangleV2.java:116)
```

The main loop had expected a singled digit integer

from zero to four, but the user typed 2.3, which the `nextInt()` was unable to process. The JVM produced an `input-mismatch-exception`. The Java 7 API for that exception is shown next.



`java.util`

Class `InputMismatchException`

`java.lang.Object`

`java.lang.Throwable`

`java.lang.Exception`

`java.lang.RuntimeException`

`java.util.NoSuchElementException`

`java.util.InputMismatchException`

All Implemented Interfaces:

`Serializable`

```
public class InputMismatchException
extends NoSuchElementException
```

Thrown by a `Scanner` to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.

Since:

1.5

When the input does not match the the scanner's expectations, the JVM makes an `input-mismatch-exception`. Your programs are expected to handle such error situations. That exception acts like a communication telling your program to handle the

error. Notice the **InputMismatchException** is an extension of **Exception**, which has an expectation that a program would normally handle such an error.

Topic: Try Catch

The try block acts like a method call. The try block shown below attempts to execute the following two commands.

```
        System.out.print("Enter 0-exit 1-setup 2-list  
3-perimeter 4-area: ");  
        choice = scan.nextInt();
```

Those commands were the source of the recent error message, when a user mistakenly entered 2.3, instead of a integer from 0 to 4. A try block for those two command appears as shown below.

```
    try  
    {  
        System.out.print("Enter 0-exit 1-setup 2-list  
3-perimeter 4-area: ");  
        choice = scan.nextInt();  
    }
```

Following that try block is an associated catch block, which catches only exceptions of class **InputMismatchException**.

Since exceptions form a family tree as shown in the API for **InputMismatchException**, programs commonly include a series of catch blocks beginning with the

lowest levels of the tree of exceptions and ending with a general catch for all exceptions as shown below.

```

    catch (InputMismatchException ime)
    {
        System.out.println("Enter only a single digit
number from 0 to 4.");
    }
    catch (Exception e)
    {
        System.out.println("Check your choice for
error.");
    }

```

The API for the **nextInt** method of the **Scanner** class lists three possible exceptions that it could throw.

nextInt

```
public int nextInt()
```

Scans the next token of the input as an int.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(rs)`.

Returns:

the int scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

Illegal state exception means the scanner is closed, which is unlikely except for a serious program error. Such an error would be detected during testing of the application because the program should not close the

scanner until the program has completed all processing.

No such element exception is more appropriate when scanning a file, not when waiting for a user to type a choice. Thus only input mismatch exception is appropriate for the test program.

The while loop requires two **nextInt** commands. Before the while loop begins the first **nextInt** gets the choice, because the top line of the while loop tests the value of the choice. At the end of the while loop the user also makes the same choice. To avoid repeating the try block and cat block the **nextInt** command was moved to a separate method.

The **getChoice** method begins by setting the value of choice to **EXIT**, so when the user's choice is invalid the program displays an error messages and ends execution. This continues the model of always starting each method by assigning a valid answer to the return variable. Thus the return statement always returns a valid answer. The value of EXIT was chosen to avoid any confusing responses to the user's choice. Also an error message is displayed informing the user of the error. Any other choice than EXIT would mean the main loop would perform another action, which would most likely not be the user's choice, thus any other choice than EXIT would confuse the user. EXIT was not chosen to punish the user.

Exiting a program at the first indication of an error frustrates the user, so an immediate exit is not the

best response. Later programs will include a loop, so the user will have multiple chances to enter a valid choice.

Also the **getChoice** method checks that the input choice is within the acceptable range of zero to four and displays an appropriate error message.

```
/**
 * getNextInt get the next int value from the scanner
 */
private int getChoice()
{
    int choice = EXIT;
    try
    {
        System.out.print("Enter 0-exit 1-setup 2-list
3-perimeter 4-area: ");
        choice = scan.nextInt();
        if (choice < EXIT || choice > AREA)
        {
            System.out.println("Your choice of " +
choice + " is not in the acceptable range of 0 ... 4");
            choice = EXIT;
        }
    }
    catch (InputMismatchException ime)
    {
        System.out.println("Enter only a single digit
number from 0 to 4.");
    }
    catch (Exception e)
    {

```

```

        System.out.println("Check your choice for
error.");
    }
    // scan.next();
    return choice;
}

/**
 * Go method
 */
private void go()
{
    int choice = EXIT;
    System.out.println("Start test of Triangle
object.");
    choice = getChoice();
    while (choice != EXIT)
    {
        switch(choice)
        {
            case EXIT:
                System.out.println("bye");
                System.exit(0);
            case MAKE:
                makeTriangle();
                break;
            case LIST:
                listTriPoints();
                break;
            case PERIMETER:
                System.out.printf("perimeter: %.3f\n",
tri.getPerimeter());
                break;

```

```

        case AREA:
            System.out.printf("area: %.3f\n",
tri.getArea());
            break;
        default:
            System.out.println("Enter 0, 1, 2, 3,
or 4");
    }
    choice = getChoice();
}

}

```

A catch block can handle more than one type of exception. The programmer simply types the list of exceptions separated by vertical bars |. For example to catch all three exceptions of the Scanner class the catch block becomes as shown below.

```

/* This catch block contains an ERROR! */
catch (InputMismatchException | NoSuchElementException
| IllegalStateException se)
{
    System.out.println("Enter only a single digit
number from 0 to 4.");
}
catch (Exception e)
{
    System.out.println("Check your choice for
error.");
}

```

Unfortunately that produces an error message because

`InputMismatchException` extends `NoSuchElementException`.

TestTriangleV3.java:100: error: Alternatives in a multi-catch statement cannot be related by subclassing catch (InputMismatchException | NoSuchElementException | IllegalStateException se)

Alternative `InputMismatchException` is a subclass of alternative `NoSuchElementException`

1 error

The API for the `Input-mismatch-exception` is shown below.

Overview	Package	Class	Use	Tree	Deprecated	Index
Prev Class	Next Class	Frames	No Frames	All Cl		
Summary: Nested Field Constr Method			Detail: Field Constr			

java.util

Class `InputMismatchException`

```

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.util.NoSuchElementException
          java.util.InputMismatchException
  
```

All Implemented Interfaces:

`Serializable`

```

public class InputMismatchException
  extends NoSuchElementException
  
```

The API shows `InputMismatchException` extends `NoSuchElementException`, so the catch block should list only `NoSuchElementException`, not both of those exceptions. The reason is that `InputMismatchException`

is also a **NoSuchElementException** because it extends **NoSuchElementException**.

By observing that family tree of exceptions the programmer understands that catching an exception also catches all exceptions listed below that exception. For example a catch **Throwable** would also catch **Throwable**, **Exception**, **RuntimeException**, **NoSuchElementException**, and **InputMismatchException**.

Notice in the program the separate catch **Exception** below the catch for **NoSuchElementException** and **InputMismatchException**. The reason for two different catch block is the different actions performed. The error messages differ, so different catch blocks are needed. Because the catch for **Exception** would catch them all, that catch must be below the specialized catch for **NoSuchElementException** and **InputMismatchException**.

The catch blocks for this program are shown below.

```
catch (NoSuchElementException | IllegalStateException  
se)  
{  
    System.out.println("Enter only a single digit number  
from 0 to 4.");  
}  
catch (Exception e)  
{  
    System.out.println("Check your choice for error.");  
}
```

Now the program prints an error message and ends the processing when the input is of the wrong type, such as 2.3 not being an `int`.

Enter 0-exit 1-setup 2-list 3-perimeter 4-area: 2.3

Enter only a single digit number from 0 to 4.

The test program is shown below.

```
import java.util.*;
import java.awt.geom.*;

/**
 * Test calculator of triangle facts
 */

public class TestTriangleV2
{
    // attributes
    Triangle tri;

    Scanner scan;

    final int EXIT = 0;
    final int MAKE = 1;
    final int LIST = 2;
    final int PERIMETER = 3;
    final int AREA = 4;

    /**
     * Default constructor
     */
    public TestTriangleV2()
```

```

{
    // initialize scanner
    scan = new Scanner(System.in);
    // Create default triangle
    Point2D.Double pointA = new Point2D.Double(0,0);
    Point2D.Double pointB = new Point2D.Double(0,1);
    Point2D.Double pointC = new Point2D.Double(1,1);
    tri = new Triangle(pointA, pointB, pointC);
}

/**
 * makePoint sets up a new Point2D.Double
 */
private Point2D.Double makePoint()
{
    double x = 0;
    double y = 0;
    Point2D.Double point = new Point2D.Double(0,0);
    System.out.println("Creating new (x,y) point");
    System.out.print("Enter x:");
    x = scan.nextDouble();
    System.out.print("Enter y:");
    y = scan.nextDouble();
    point = new Point2D.Double(x,y);
    return point;
}

/**
 * makeTriangle sets up a new triangle.
 */
private void makeTriangle()
{
    Point2D.Double a = makePoint();

```

```

    Point2D.Double b = makePoint();
    Point2D.Double c = makePoint();
    tri = new Triangle(a,b,c);
}

/**
 * listTriPoints displays the (x,y) coordinates of
all three vertices
 */
private void listTriPoints()
{
    Point2D.Double a = tri.getPointA();
    Point2D.Double b = tri.getPointB();
    Point2D.Double c = tri.getPointC();
    System.out.println("Triangle");
    System.out.printf("Vertex\tx\ty\n");
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'A',
a.getX(), a.getY());
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'B',
b.getX(), b.getY());
    System.out.printf("%c\t%5.3f\t%5.3f\n", 'C',
c.getX(), c.getY());
}

/**
 * getNextInt get the next int value from the scanner
 */
private int getChoice()
{
    int choice = EXIT;
    try
    {
        System.out.print("Enter 0-exit 1-setup 2-list

```

```

3-perimeter 4-area: ");
        choice = scan.nextInt();
        if (choice < EXIT || choice > AREA)
        {
            System.out.println("Your choice of " +
choice + " is not in the acceptable range of 0 ... 4");
            choice = EXIT;
        }
    }
    catch (NoSuchElementException |
IllegalStateException se)
    {
        System.out.println("Enter only a single digit
number from 0 to 4.");
    }
    catch (Exception e)
    {
        System.out.println("Check your choice for
error.");
    }
    return choice;
}

/**
 * Go method
 */
private void go()
{
    int choice = EXIT;
    System.out.println("Start test of Triangle
object.");
    choice = getChoice();
    while (choice != EXIT)

```

```

{
    switch(choice)
    {
        case EXIT:
            System.out.println("bye");
            System.exit(0);
        case MAKE:
            makeTriangle();
            break;
        case LIST:
            listTriPoints();
            break;
        case PERIMETER:
            System.out.printf("perimeter: %.3f\n",
tri.getPerimeter());
            break;
        case AREA:
            System.out.printf("area: %.3f\n",
tri.getArea());
            break;
        default:
            System.out.println("Enter 0, 1, 2, 3,
or 4");
    }
    choice = getChoice();
}

}

/**
 * Main method
 */
public static void main(String[] args)

```

```
{  
    System.out.println("Start test run of  
calculator.");  
    TestTriangleV2 obj = new TestTriangleV2();  
    obj.go();  
}  
  
}
```

Chapter Eight: event-driven programming

Topic: GUI and Event-driven programming

In this chapter your Java computer program will allow users to interact with the graphical display. Java provides the **Jbutton** class to produce buttons, the **JList** class for lists, the **JTextField** for text fields, **JCheckBox** for a check box and **JMenuItem** for a menu item. Those items can be added to a GUI object such as a **JPanel** object. For example this simple Java GUI program has a single object, a button.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ChameleonBtn extends JFrame
{
    public ChameleonBtn()
    {
        JPanel panel = new JPanel();
        final JButton btn = new JButton("change color");
        panel.add(btn);
        // A button listener is needed.
        add(panel);
    }

    public static void main(String[] args)
    {
        JFrame frame = new ChameleonBtn();
        frame.setTitle("chameleon");
    }
}
```

```

        frame.setLocationRelativeTo(null);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(820, 800);
        frame.pack();
        frame.setVisible(true);
    }

}

```

Class **ChameleonBtn** extends **JFrame** so instances of **ChameleonBtn** produce a GUI interface with a frame. You can set the title of the frame, its locations, and size. Also **setDefaultCloseOperation** with parameter value of **JFrame.EXIT_ON_CLOSE** allows the frame close button labeled X to work properly. The final step is **frame.setVisible(true)**; Additional changes made after that step might not be included in the GUI display.

The constructor for class **ChameleonBtn** creates an instance of **JButton** which is attached to an instance of **JPanel**. Finally that panel is attached to the frame by a simple **add** command. Pressing the change color button produces no result. Next our Java program must include a listener to react to the **ActionEvent** resulting from pressing that button.

Every time a user uses the mouse to press a button, select from a list of choices, or move the scroll bar that user action is an event. Likewise pressing a key is an event. When the user interacts with a source object or source component that object or component

produces an event. When you select from a **JList** object a **ListSelectionEvent** is produced.

All event types are extensions of the core **EventObject**. The **AWTEvent**, **ListSelectionEvent**, and **ChangeEvent** are direct extensions of the **EventObject**. **ActionEvent**, **AdjustEvent**, **ComponentEvent**, **ItemEvent**, and **TextEvent** are extensions of the **AWTEvent** object. The **ComponentEvent** continues to support extensions for the **ContainerEvent**, **FocusEvent**, **InputEvent**, **PaintEvent**, and **WindowEvent**. Finally the **InputEvent** is extended by the **MouseEvent** and **KeyEvent** objects. Thus an entire family of event objects has been developed for each possible interaction with a Java GUI program. The core **EventObject** provides **getSource** and **toString** methods. When one listener is attached to several instances of **JButton** objects then the listeners commonly use **getSource** to identify which button was pushed.

When you press a **Jbutton** an **ActionEvent** is produced. Your java computer program must create an action listener and register that action listener with the button by using the **addActionListener** method. The **ActionListener** interface requires implementation of the **actionPerformed** method.

Stage 1: use a listener to display a message on the standard output stream

The implementation included here simply displays a

message that the button was pushed onto system output.

```
public void actionPerformed(ActionEvent e)
{
    System.out.println("Button was pushed");
}
```

The ButtonListener is in a separate class.

```
import java.awt.event.*;

public class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Button was pushed");
    }
}
```

Now the **PushMeBtn** class can create an instance of **ButtonListener** and use the **addActionListener** method to register that button listener with the button.

```
ButtonListener btnListener = new
ButtonListener();
btn.addActionListener( btnListener );
```

The main class demonstrates the role of the button producing an event which will be handled by the button listener.

```
import java.awt.*;
import javax.swing.*;
```

```
import java.awt.event.*;

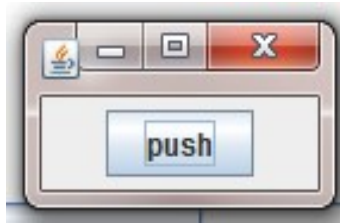
public class PushMeBtn extends JFrame
{
    public PushMeBtn()
    {
        JPanel panel = new JPanel();
        final JButton btn = new JButton("push");
        panel.add(btn);
        ButtonListener btnListener = new
ButtonListener();
        btn.addActionListener( btnListener );
        add(panel);
    }

    public static void main(String[] args)
    {
        JFrame frame = new PushMeBtn();
        frame.setTitle("push me");
        frame.setLocationRelativeTo(null);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(820, 800);
        frame.pack();
        frame.setVisible(true);
    }

}
```

The java computer program produces this GUI.



Pressing the button produced this statement on system output.

```
----jGRASP exec: java -ea -ea PushMeBtn

Button was pushed
,
```

Stage 2: share a parameter to change the button's color

A GUI program does not normally print on the standard output stream. Instead results are displayed on the screen. For our next iteration of this program the button should simply change color. Now the **actionPerformed** method needs access to that button. The button is defined in a different class, so the button must be passed as an argument to the **ButtonListener** class. The **actionPerformed** method is defined by the **ActionListener** interface, so our computer program cannot change that method's signature to include the button. Instead a new constructor for the **ButtonListener** will accept the button and save it as the value of a new attribute named **btn**.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

```
/**
 * implements interface ActionListener
 */

public class ButtonListener implements ActionListener
{

    // attribute
    JButton btn;

    /**
     * fully parametrized constructor
     *
     * @param JButton button
     */
    public ButtonListener(JButton _btn)
    {
        btn = _btn;
    }

    /**
     * interface ActionListener requires implementation
    of this method
     *
     * @see java.awt.event.ActionListener
     */
    public void actionPerformed(ActionEvent e)
    {
        btn.setBackground(Color.ORANGE) ;
    }

}
```

The test program is also changed to include the button in the call to construct a new instance of `ButtonListener`. Instead of `new ButtonListener()` the command is `new ButtonListener(btn)`.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ChameleonBtn extends JFrame
{
    public ChameleonBtn()
    {
        JPanel panel = new JPanel();
        final JButton btn = new JButton("push");
        panel.add(btn);
        ButtonListener btnListener = new
ButtonListener(btn);
        btn.addActionListener( btnListener );
        add(panel);
    }

    public static void main(String[] args)
    {
        JFrame frame = new ChameleonBtn();
        frame.setTitle("chameleon");
        frame.setLocationRelativeTo(null);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(820, 800);
        frame.pack();
        frame.setVisible(true);
    }
}
```

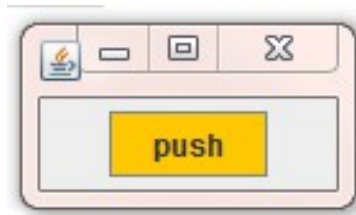
```

    }

}

```

When the button is pressed it changes color to orange.



Stage 3: use an inner class for the listener

A further refactoring of this program is to move the listener class into the test class. A class within a class is known as an inner class. The main reason for this refactoring is that an inner class automatically is aware of the attributes of the surrounding class. For our program the listener as an inner class would automatically be aware of the button as an attribute. Then the button would not need to be passed as an argument to the listener. Be aware that the button must become an attribute.

```

import java.awt.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class InnerButtonListener extends JFrame
{
    private JButton btn;

```

```
public InnerButtonListener()
{
    JPanel panel = new JPanel();
    btn = new JButton("push");
    panel.add(btn);
    ButtonListener btnListener = new
ButtonListener();
    btn.addActionListener( btnListener );
    add(panel);
}

public static void main(String[] args)
{
    JFrame frame = new InnerButtonListener();
    frame.setTitle("chameleon");
    frame.setLocationRelativeTo(null);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(820, 800);
    frame.pack();
    frame.setVisible(true);
}

/**
 * inner class button listener implements interface
ActionListener
 */

public class ButtonListener implements
ActionListener
{

```

```

    /**
     * interface ActionListener requires
implementation of this method
     *
     * @see java.awt.event.ActionListener
     */
    public void actionPerformed(ActionEvent e)
    {
        btn.setBackground(Color.ORANGE);
    }

}

}

```

Step 4: Use an anonymous inner class for the button listener

Since the button listener is only used with the button programmers often use an anonymous inner class for the listener. Then the `btn.addActionListener()` method begins with the `new ActionListener()` method followed by a class definition with the curly braces `{}`. That class definition does not include any declaration such as `public class Anonymous`. Instead the class definition simply includes the methods. The **ActionListener** interface requires the `actionPerformed` method to be declared so the anonymous class includes only that method which sets the background color of the button to orange.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

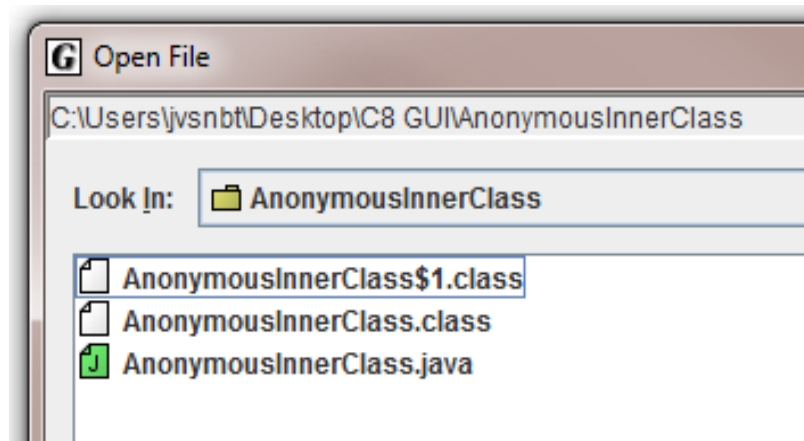
public class AnonymousInnerClass extends JFrame
{
    public AnonymousInnerClass()
    {
        JPanel panel = new JPanel();
        final JButton btn = new JButton("push");
        panel.add(btn);
        btn.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent
e)
                {
                    btn.setBackground(Color.ORANGE);
                }
            } );
        add(panel);
    }

    public static void main(String[] args)
    {
        JFrame frame = new AnonymousInnerClass();
        frame.setTitle("chameleon");
        frame.setLocationRelativeTo(null);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(820, 800);
        frame.pack();
        frame.setVisible(true);
    }
}
```

```
}
```

When you examine the directory containing your java computer program you will notice your java file, the associated class file, and another class file ending with \$1. Each inner class has \$number attached with the number increasing from 1 for each inner class.



Anonymous inner classes are popular because the action performed method is closely associated with the button or other GUI object. Also the coding is briefer.

Topic: Layout managers

A layout manager controls the placement of objects on the screen. Java provides several layout managers such as **FlowLayout**, **GridLayout**, and **BorderLayout**, all of which will be used in developing this application for studying triangles.

Topic: FlowLayout

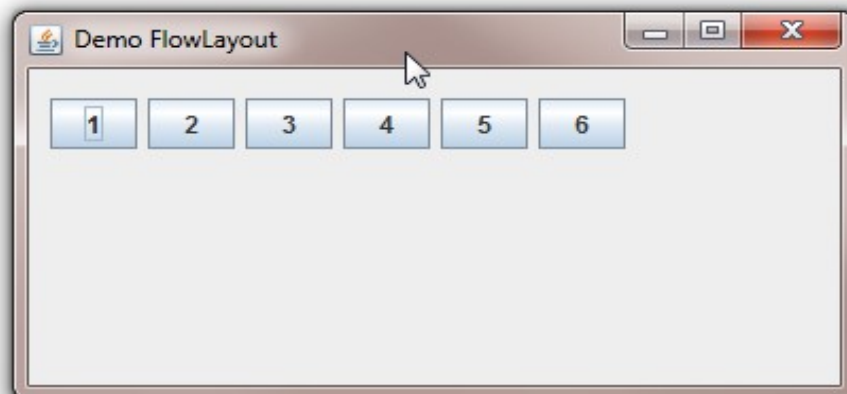
The **setLayout** command sets the layout manager for the application. Until now our graphics programs used a `Jpanel` which has the default **FlowLayout** manager, the simplest layout manager. Being the default our java programs have not included the **setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20))** command which creates an anonymous instance of **FlowLayout** having the command to align objects to the left with a horizontal gap between objects of 10 pixels and a vertical gap between objects of 20 pixels. **FlowLayout** sets the objects from left to right across the screen. When a row is full **FlowLayout** advances down to the next row. **LEFT** is a static constant of the **FlowLayout** class. This simple java program will demonstrate the use of flow layout.

```
import javax.swing.*;
import java.awt.*;

public class DemoFlowLayout extends JFrame
{
    /**
     * constructor
     */
    public DemoFlowLayout()
    {
        setTitle("Demo FlowLayout");
        setSize(200, 200);
        setLocationRelativeTo(null); // Center the frame
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new FlowLayout(FlowLayout.LEFT, 5,
```

```
10));  
    JPanel panel = new JPanel();  
    panel.setLayout(new  
FlowLayout(FlowLayout.LEFT,5,5));  
    panel.add(new JButton("1"));  
    panel.add(new JButton("2"));  
    panel.add(new JButton("3"));  
    panel.add(new JButton("4"));  
    panel.add(new JButton("5"));  
    panel.add(new JButton("6"));  
    add(panel);  
    setVisible(true);  
}  
  
/**  
 * main method  
 */  
public static void main(String[] args)  
{  
    DemoFlowLayout demo = new DemoFlowLayout();  
}  
  
}
```



Topic: BorderLayout

The **BorderLayout** layout manager divides the screen into five areas: East, South, West, North, and Center. To add a new button to the east side of the screen the command would be **add(new JButton("East"), BorderLayout.EAST);** The component requiring the most space is usually placed in the center. This java program demonstrates the use of border layout.

```
import javax.swing.*;
import java.awt.*;

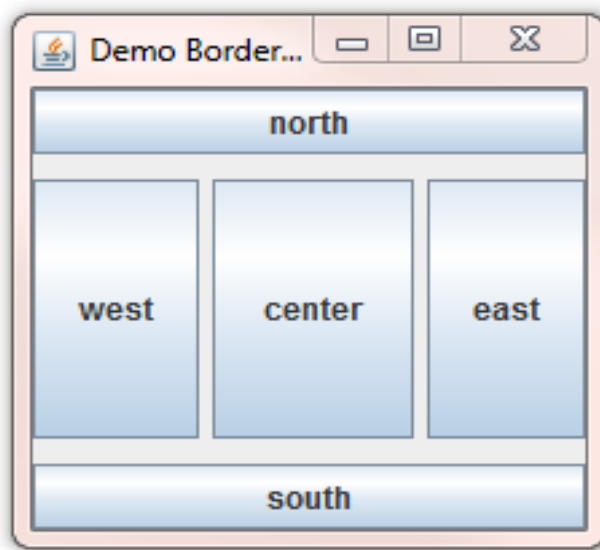
public class DemoBorderLayout extends JFrame
{

    /**
     * constructor
     */
    public DemoBorderLayout()
    {
        setTitle("Demo Border Layout");
        setSize(200, 200);
        setLocationRelativeTo(null); // Center the frame
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Set BorderLayout with horizontal gap 5 and
vertical gap 10
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout(5, 10));
        panel.add(new JButton("center"),
BorderLayout.CENTER);
        panel.add(new JButton("east"),
BorderLayout.EAST);
```

```
        panel.add(new JButton("west"),
BorderLayout.WEST);
        panel.add(new JButton("north"),
BorderLayout.NORTH);
        panel.add(new JButton("south"),
BorderLayout.SOUTH);
        add(panel);
        setVisible(true);
    }

    /**
     * main method
     */
    public static void main(String[] args)
    {
        DemoBorderLayout demo = new DemoBorderLayout();
    }
}
```



Topic: GridLayout

The **GridLayoutManager** divides the container into a rectangular grid. The command **setLayout(new GridLayout(3,2,5,5))** will divide the container into a grid of three rows each containing two columns. Both the horizontal and vertical separation between objects will be ten pixels.

This example java program will demonstrate the use of grid layout.

```
import javax.swing.*;
import java.awt.*;

public class DemoGridLayout extends JFrame
{

    /**
     * constructor
     */
    public DemoGridLayout()
    {
        setTitle("Demo GridLayout");
        setSize(200, 200);
        setLocationRelativeTo(null); // Center the frame
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

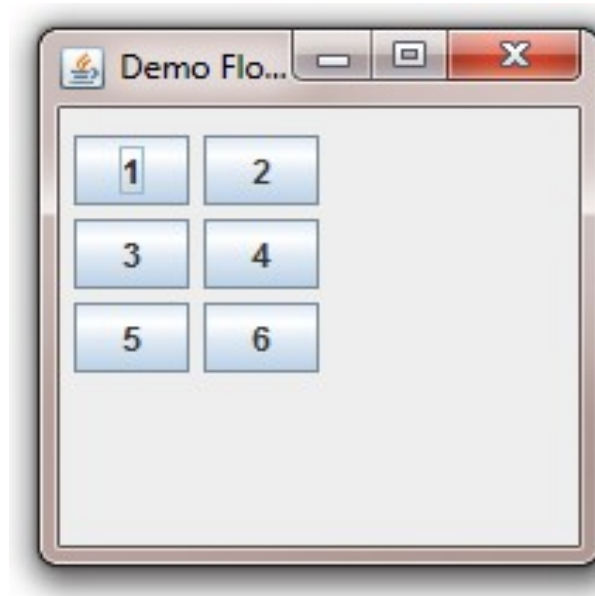
        setLayout(new FlowLayout(FlowLayout.LEFT, 5,
10));

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(3,2,5,5));
        panel.add(new JButton("1"));
        panel.add(new JButton("2"));
```

```
panel.add(new JButton("3"));
panel.add(new JButton("4"));
panel.add(new JButton("5"));
panel.add(new JButton("6"));
add(panel);
setVisible(true);
}

/**
 * main method
 */
public static void main(String[] args)
{
    DemoGridLayout demo = new DemoGridLayout();
}

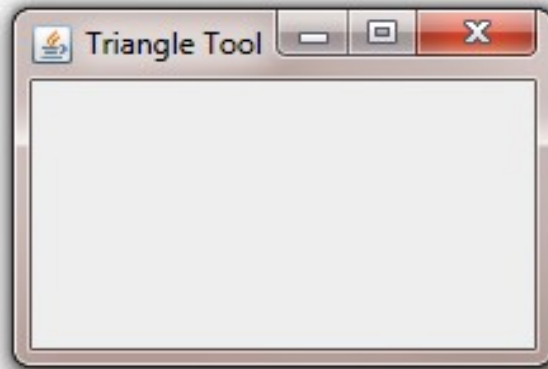
}
```



Topic: Event-driven GUI programming

Next we will use our new knowledge of event-driven

programming to develop a GUI program to draw and study triangles. Development of the program begins with an empty **JFrame**.



The corresponding java computer program uses a class that extends **JFrame** so the class definition begins with **public class TriangleToolV1 extends JFrame** followed by the body of the class within curly braces **{}** also known as curly brackets. **JFrame** requires an import statement for **java.swing.JFrame** or more generally **import java.swing.*;**. A **main** method is required to begin execution of your GUI program. The constructor has a **setTitle("Triangle Tool")** that sets the frame's title. Next the **setSize(800,600)** command sets the frame's size and the command **setLocationRelativeTo(null)** sets the frame's location.

The layout is set to **BorderLayout**. Because the screen area for displaying a triangle would demand the most space that screen area can be set in the center and command the largest share of the screen's space.

```
import javax.swing.*;  
import java.awt.*;
```

```
/**
 * Tool for working with triangles
 */
public class TriangleToolV1 extends JFrame
{

    /**
     * default constructor for triangle tool
     */
    public TriangleToolV1()
    {
        // Set up GUI
        setTitle("Triangle Tool");
        setSize(800, 600);
        setLocationRelativeTo(null); // Center the frame
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Set BorderLayout with horizontal gap 5 and
vertical gap 10
        setLayout(new BorderLayout(5, 10));
        setVisible(true);
    }

    /**
     * starts the program
     */
    private void go()
    {

    }

    /**
```

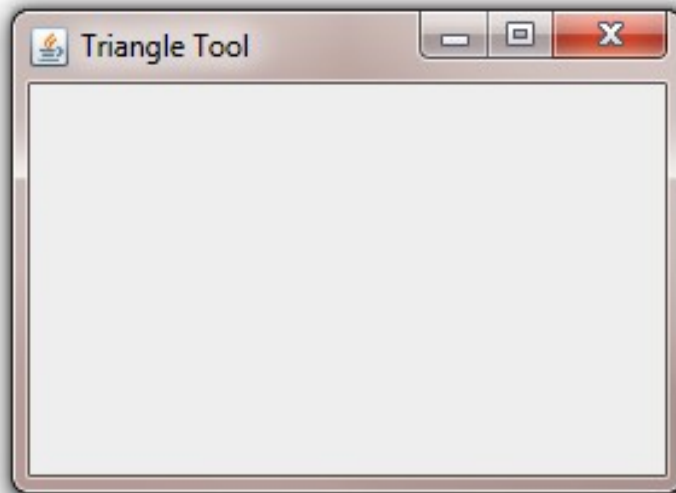
```

    * main method
    */
    public static void main(String[] args)
    {
        TriangleToolV1 tool = new TriangleToolV1();
        tool.go();
    }

} // end class TriangleTool

```

The program produces an empty frame.



For the next step of this application's development our program adds a panel to the center region of the screen. An inner class named **SketchPanel** extends **JPanel** so the **paintComponent** method can be overwritten with our own code to draw triangles.

```

/**
    * inner class SketchPanel for drawing triangles,

```

```

etc
    */
    class SketchPanel extends JPanel
    {

        /**
         * Override JPanel's paintComponent method
         *
         * @param Graphics object g
         */
        protected void paintComponent(Graphics g)
        {
            super.paintComponent(g);
            g.drawString("triangles will be drawn here",
0, 40);
        }
    }

```

The GUI update method within the java virtual machine automatically calls this **paintComponent** method and supplies the graphics object named **g**. The **super.paintComponent(g)** command calls the **paintComponent** method of the superclass, or parent class, so the screen area will be erased and made ready for our new graphics command. Instead of drawing a triangle this program simply draws a string stating "triangles will be drawn here".

Next the Triangle tool's constructor will create an instance of the sketch pad class and add that new panel to the center of the frame.

```
// Add sketch pad to draw triangle
```

```
JPanel sketchPad = new SketchPanel();  
sketchPad.add(new JLabel("sketch pad"));  
add(sketchPad, BorderLayout.CENTER);
```

The new label named "sketch pad" temporarily shows that the panel has been added. That label will be removed later.

The entire program is here.

```
import javax.swing.*;  
import java.awt.*;  
  
/**  
 * Tool for working with triangles  
 */  
public class TriangleToolV2 extends JFrame  
{  
  
    /**  
     * default constructor for triangle tool  
     */  
    public TriangleToolV2()  
    {  
        // Set up GUI  
        setTitle("Triangle Tool");  
        setSize(800, 600);  
        setLocationRelativeTo(null); // Center the frame  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        //Set BorderLayout with horizontal gap 5 and  
        vertical gap 10
```

```
setLayout(new BorderLayout(5, 10));

// Add sketch pad to draw triangle
JPanel sketchPad = new SketchPanel();
sketchPad.add(new JLabel("sketch pad"));
add(sketchPad, BorderLayout.CENTER);

setVisible(true);
}

/**
 * starts the program
 */
private void go()
{

}

/**
 * main method
 */
public static void main(String[] args)
{
    TriangleToolV2 tool = new TriangleToolV2();
    tool.go();
}

/**
 * inner class SketchPanel for drawing triangles,
etc
 */
class SketchPanel extends JPanel
{
```

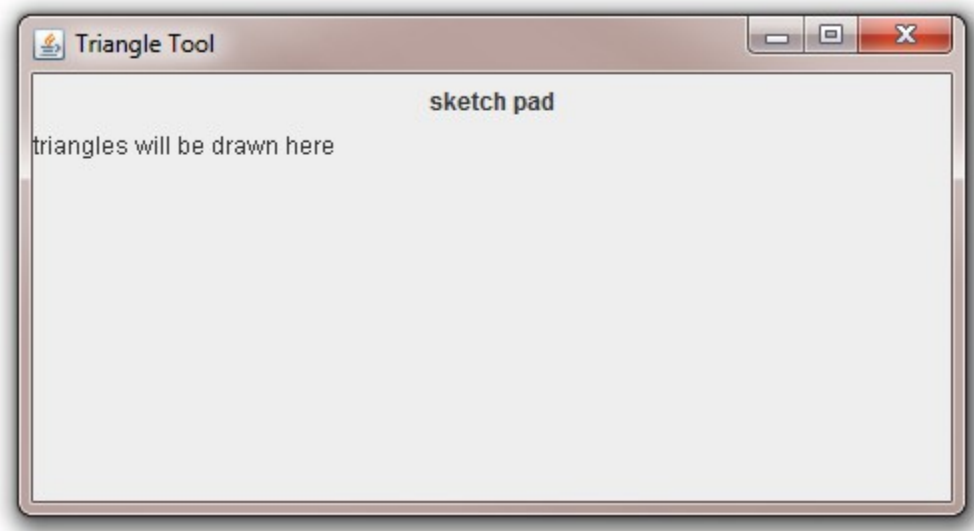
```

/**
 * Override JPanel's paintComponent method
 *
 * @param Graphics object g
 */
protected void paintComponent(Graphics g)
{
    super.paintComponent(g);
    g.drawString("triangles will be drawn here",
0, 40);
}

} // end class TriangleTool

```

The triangle application now displays the label "sketch pad" and the message "triangles will be drawn here".



Of course the next step is to draw a triangle. That

requires establishing a model of the triangle. The easiest model is to keep the x and y coordinates of the three vertices of the triangle. For that purpose a separate class named **TriangleCartesian** will store the x and y coordinates of each vertex as `Point2D.Double` objects.

As an aside a common paradigm for developing GUI programs is named model-view-controller and described at

http://en.wikipedia.org/wiki/Model_view_controller .

Although this program uses a variable named model to represent the triangle, this application is not an example of the model-view-controller paradigm.

```
import java.awt.geom.*;
import java.awt.*;

/**
 * Class CartesianTriangle uses three Points to
 * describe its triangle.
 */

public class TriangleCartesian
{

    // Attributes
    Point2D.Double pointA, pointB, pointC;

    // Constructors

    /**
     * default constructor
```

```

    */
    public TriangleCartesian()
    {
        pointA = new Point2D.Double(); // by default set
to (0,0)
        pointB = new Point2D.Double();
        pointC = new Point2D.Double();
    }

    /**
     * fully parametrized constructor
     *
     * @param Point2D.Double three vertices of triangle
     */
    public TriangleCartesian(Point2D.Double a,
Point2D.Double b, Point2D.Double c)
    {
        pointA = a;
        pointB = b;
        pointC = c;
    }

    // Accessors and mutators

    /**
     * getPointA returns the Point object for attribute
pointA
     *
     * @return Point2D.Double for vertex A
     */
    public Point2D.Double getPointA()
    {
        return pointA;
    }

```

```

    }

    /**
     * setPointA sets pointA to the Point provided in
the list of arguments
     *
     * @param the point for this vertex
     */
    public void setPointA(Point2D.Double point)
    {
        pointA = point;
    }

    /**
     * getPointB returns the Point object for attribute
pointB
     *
     * @return vertex B
     */
    public Point2D.Double getPointB()
    {
        return pointB;
    }

    /**
     * getPointB sets pointB to the Point provided in
the list of arguments
     *
     * @param the point for this vertex
     */
    public void setPointB(Point2D.Double point)
    {
        pointB = point;
    }

```

```

    }

    /**
     * getPointC returns the Point object for vertex
pointC
     *
     * @return vertex C
     */
    public Point2D.Double getPointC()
    {
        return pointC;
    }

    /**
     * getPointA sets pointA to the Point provided in
the list of arguments
     *
     * @param the point for this vertex
     */
    public void setPointC(Point2D.Double point)
    {
        pointC = point;
    }

    /**
     * get length of side A - B
     */
    public double getSideAB()
    {
        // Point2D.Double extends Point2D which provides
the distance method
        return pointA.distance(pointB) ;
    }

```

```
/**
 * get length of side B-C
 *
 * @return length of side BC
 */
public double getSideBC()
{
    return pointB.distance(pointC);
}

/**
 * get length of side A - C
 *
 * @return length of side AC
 */
public double getSideAC()
{
    return pointA.distance(pointC);
}

/**
 * get perimeter of triangle
 *
 * @return perimeter
 */
public double getPerimeter()
{
    return getSideAB() + getSideBC() + getSideAC();
}

/**
 * calculate area of triangle
```

* In geometry, Heron's (or Hero's) formula, named after Heron of Alexandria,

* states that the area T of a triangle whose sides have lengths a , b , and c is

* $T = \sqrt{s(s-a)(s-b)(s-c)}$

* where s is the semiperimeter of the triangle:

*

* @return area

*/

```
public double getArea()
```

```
{
```

```
    double s = getPerimeter() / 2;
```

```
    double a = getSideAB();
```

```
    double b = getSideBC();
```

```
    double c = getSideAC();
```

```
    double area = Math.sqrt(s * (s-a) * (s-b) * (s-c));
```

```
    return area;
```

```
}
```

```
/**
```

* drawTriangle uses a graphics object g to draw the triangle

*

* @param graphics object

*

* @see java.awt.Polygon

*/

```
public void drawTriangle(Graphics g)
```

```
{
```

```
    final int SCALE = 10;    // Scale by 10 then truncate to int value for locations
```

```
    Polygon poly = new Polygon();
```

```

        poly.addPoint((int) (SCALE * pointA.getX()),
(int) (SCALE * pointA.getY()));
        poly.addPoint((int) (SCALE * pointB.getX()),
(int) (SCALE * pointB.getY()));
        poly.addPoint((int) (SCALE * pointC.getX()),
(int) (SCALE * pointC.getY()));
        g.drawPolygon(poly);
    }

}

```

Notice that the model stores the x and y coordinates as **double** values, but the draw method truncates those double values to **int** values before drawing the triangle. Also class **Polygon** provides a draw method which this class wisely employs. The strength of java is the immense collection of classes provided for our use.

Since java's coordinate system uses only the quadrant of the Cartesian space where both x and y coordinates have positive values our triangle will be required to appear in that quadrant. Also (0,0) is located in the top-left corner. The X coordinate increases from left to right, but the Y coordinate increases from top to bottom, so our triangle will display upside down.

Developing a more realistic coordinate system is a different problem left to the reader.

The triangle tool must define the triangle model, named **triangle**, as an attribute so the paint method of the inner class will have access to that model. The

command to create that model uses class Triangle Cartesian which require three double valued points.

```
// Make model of triangle using vertices.
triangle = new TriangleCartesian(
    new Point2D.Double(0.0, 0.0),
    new Point2D.Double(22.3, 34.9),
    new Point2D.Double(35.7, 50.2)
);
```

The draw component method of the sketch panel must now draw that triangle. The additional command is simply **triangle.drawTriangle(g)**.

The main class now includes the following commands.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

/**
 * Tool for working with triangles
 */
public class TriangleToolV3 extends JFrame
{
    TriangleCartesian triangle;

    /**
     * default constructor for triangle tool
     */
    public TriangleToolV3()
    {
        // Set up GUI
        setTitle("Triangle Tool");
    }
}
```

```
setSize(800, 600);
setLocationRelativeTo(null); // Center the frame
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Set BorderLayout with horizontal gap 5 and
vertical gap 10
setLayout(new BorderLayout(5, 10));

// Add sketch pad to draw triangle
JPanel sketchPad = new SketchPanel();
// sketchPad.add(new JLabel("sketch pad"));
add(sketchPad, BorderLayout.CENTER);

// Make model of triangle using vertices.
triangle = new TriangleCartesian(
    new Point2D.Double(0.0, 0.0),
    new Point2D.Double(22.3, 34.9),
    new Point2D.Double(35.7, 50.2)
);

setVisible(true);
}

/**
 * starts the program
 */
private void go()
{

}

/**
```

```

    * main method
    */
    public static void main(String[] args)
    {
        TriangleToolV3 tool = new TriangleToolV3();
        tool.go();
    }

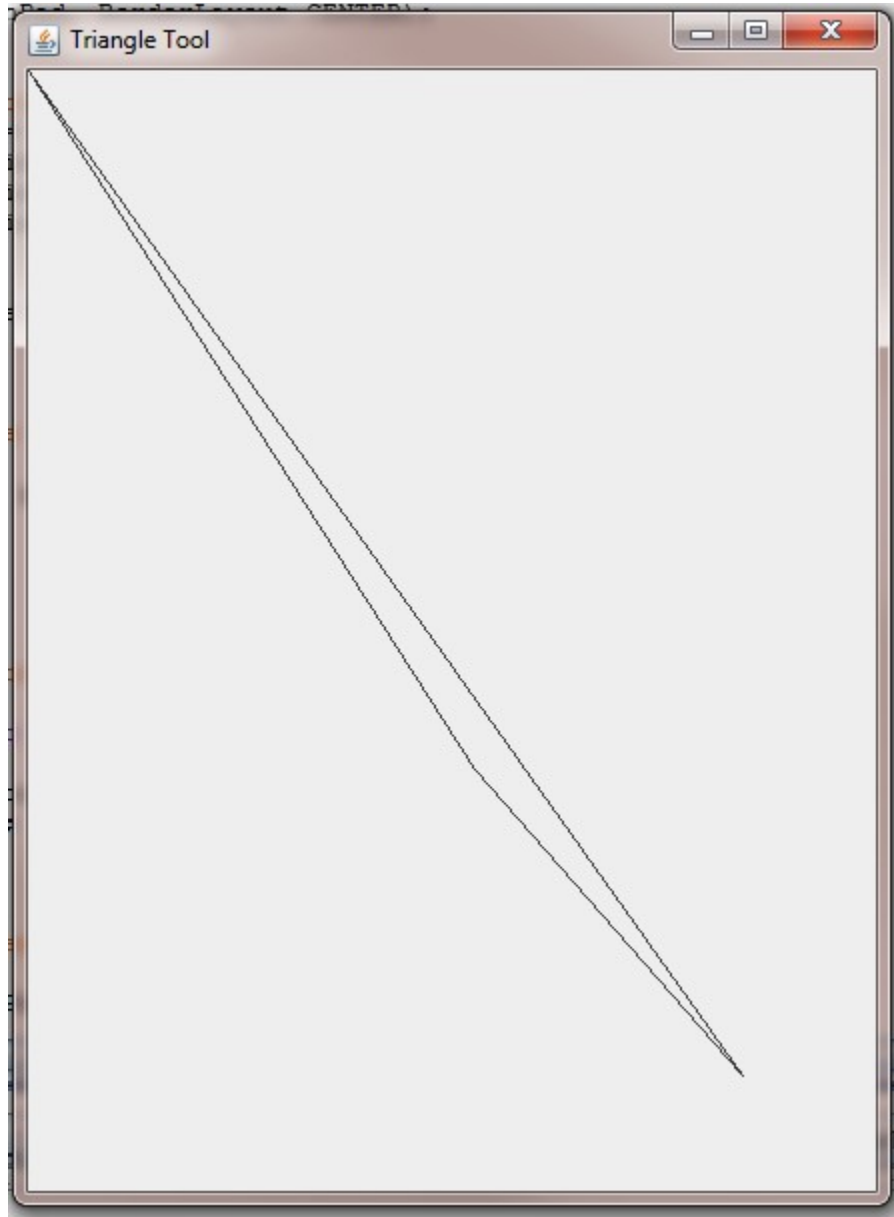
    /**
     * inner class SketchPanel for drawing triangles,
    etc
     */
    class SketchPanel extends JPanel
    {

        /**
         * Override JPanel's paintComponent method
         *
         * @param Graphics object g
         */
        protected void paintComponent(Graphics g)
        {
            super.paintComponent(g);
            triangle.drawTriangle(g);
        }
    }

} // end class TriangleTool

```

The program now draws a triangle as shown here.



Also the triangle program must create a panel for the button to draw the triangle. The action performed method of that anonymous listener simply call repaint. That method calls the paint component method which is redefined to draw the triangle. The additional command is simply **triangle.drawTriangle(g)**.

The next enhancement will create a data panel for the user to enter coordinates for all three vertices and

to view calculated results such as area and length of the perimeter.

```

        // Create data panel for points and results at
right (EAST) side
        dataPanel = new JPanel();
        dataPanel.setLayout(new BorderLayout(5, 10));

        // Add x,y addresses of three defining points of
triangle
        TriangleTfGrid = new JPanel();
        TriangleTfGrid.setLayout(new GridLayout(4, 2, 5,
5));
        TriangleTfGrid.add(new JLabel("x"));
        TriangleTfGrid.add(new JLabel("y"));
        // Add 2 x 3 array of JTextField
        trianglePointsTf = new ArrayList<PointXYTf>();

        for (int i = 0; i < VERTEX_CNT; i++)
        {
            PointXYTf ptTf = new PointXYTf();
            trianglePointsTf.add(ptTf);
            TriangleTfGrid.add(ptTf.getTfX());
            TriangleTfGrid.add(ptTf.getTfY());
        }
        dataPanel.add(TriangleTfGrid,
BorderLayout.NORTH);

```

The class named PointTf holds the text fields for the x and y coordinates of a single vertex of the triangle. Since the pair of x and y coordinates for a vertex represent the coordinates of an object they should be themselves stored in an object. All those

rows of text fields are stored in an array list named trianglePointsTf.

The PointTf class is listed here.

```
import javax.swing.*;
import java.awt.geom.*;

/**
 * Maintain textFields for the x and y coordinates of a
 * point
 * Always maintain the point and the textfields with
 * equivalent values
 */
public class PointXYTf
{
    JTextField tfX, tfY;
    final int JTF_WIDTH = 8;
    int jtfWidth;
    final String JTF_INIT_VALUE = "0.0";

    /**
     * Default constructor
     */
    public PointXYTf()
    {
        jtfWidth = JTF_WIDTH;

        tfX = new JTextField();
        tfX.setText(JTF_INIT_VALUE);

        tfY = new JTextField();
        tfY.setText(JTF_INIT_VALUE);
    }
}
```

```

}

/**
 * fully parametrized constructor using JTextFields
 */
public PointXYTf(JTextField tfx, JTextField tfy)
{
    tfX = tfx;
    tfY = tfy;
}

/**
 * fully parametrized constructor using Point2D.Double
 *
 * @param Point2D.Double the point matching these
text fields
 */
public PointXYTf(Point2D.Double pt)
{
    // point = pt;
    double x = pt.getX();
    String xStr = new Double(x).toString();
    tfX = new JTextField(xStr);
    double y = pt.getY();
    String yStr = new Double(y).toString();
    tfY = new JTextField(yStr);
}

/**
 * Set contents of tfX and tfY for point
 *
 * @param Point2D.Double point matching these text
fields

```

```
*/  
  
public void setTfXY(Point2D.Double p)  
{  
    double x = p.getX();  
    Double dX = new Double(x);  
    String xString = dX.toString();  
    tfX.setText(xString);  
    double y = p.getY();  
    Double dY = new Double(y);  
    String yString = dY.toString();  
    tfY.setText(yString);  
}  
  
/**  
 * Set x value  
 *  
 * @param double x coordinate  
 */  
public void setTfX(double x)  
{  
    Double dX = new Double(x);  
    String xString = dX.toString();  
    tfX.setText(xString);  
}  
  
/**  
 * Set y value  
 *  
 * @param double y coordinate  
 */  
public void setTfY(double y)  
{  
    Double dY = new Double(y);
```

```
String yString = dY.toString();
tfY.setText(yString);
}

/**
 * Get textfield X
 *
 * @return JTextField for X coordinate
 */
public JTextField getTfX()
{
    return tfX;
}

/**
 * Get textfield Y
 *
 * @return JTextField for Y coordinate
 */
public JTextField getTfY()
{
    return tfY;
}

/**
 * return Point2D.Double
 *
 * @return Point2D with X,Y coordinates
 */
public Point2D.Double getPoint2D()
{
    String strX = tfX.getText();
    double x = new Double(strX).doubleValue();
```

```

String strY = tfY.getText();
double y = new Double(strY).doubleValue();
Point2D.Double point = new Point2D.Double(x,y);
return point;
}

/**
 * toString method
 *
 * @return description of x,y coordinates
 */
public String toString()
{
    return "[" + tfX.getText() + "," + tfY.getText()
+ "]" ;
}

}

```

Notice the PointXYTf class can return the vertex as a Point 2D.Double object which satisfies the requirements of the Polygon class. So these classes cooperate well.

The results include area and perimeter. A new grid layout holds the text fields for those results.

```

// Add 2 x 2 array of calculated results
results = new JPanel();
results.setLayout(new GridLayout(2, 2, 5, 5));
results.add(new JLabel("area"));
jtfArea = new JTextField(8);
jtfArea.setText("0.0");
area = 0;

```

```

results.add(jtfArea);
results.add(new JLabel("perimeter"));
jtfPerimeter = new JTextField(8);
jtfPerimeter.setText("0.0");
results.add(jtfPerimeter);
perimeter = 0;
dataPanel.add(results, BorderLayout.SOUTH);

```

That data panel is now added to the surrounding frame using this command `add(dataPanel, BorderLayout.EAST);`

A control panel is created for the button named `calculate`, which will both perform the calculations and draw the new triangle.

```

// Add control panel to the bottom (SOUTH) of
the window

controls = new JPanel();
controls.setLayout(new
FlowLayout(FlowLayout.LEFT, 10, 20));
jbtnCalc = new JButton("calculate");
jbtnCalc.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            calcTriangle();
            repaint();
        }
    }
);
controls.add(jbtnCalc);
add(controls, BorderLayout.SOUTH);

```

Finally the method to perform the calculations, named `calcTriangle`, must be written. It gets the vertex from each row of the table of vertices. Those

The final program is shown here.

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.awt.geom.*;
5  import java.util.*;
6
7  /**
8   * Tool for working with triangles
9   */
10 public class TriangleToolV4 extends JFrame
11 {
12     TriangleCartesian triangle;
13     ArrayList<PointXYTf> trianglePointsTf;
14     JTextField jtfArea, jtfPerimeter;
15     double area, perimeter;
16
17     /**
18      * default constructor for triangle tool
19      */
20     public TriangleToolV4()
21     {
22         // Set up GUI
23         setTitle("Triangle Tool");
24         setSize(800, 600);
25         setLocationRelativeTo(null); // Center the
frame

```

```

26
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27      //Set BorderLayout with horizontal gap 5 and
vertical gap 10
28      setLayout(new BorderLayout(5, 10));
29
30      // Add sketch pad to draw triangle
31      JPanel sketchPad = new SketchPanel();
32      // sketchPad.add(new JLabel("sketch pad"));
33      add(sketchPad, BorderLayout.CENTER);
34
35      // Create data panel for points and results
at right (EAST) side
36      JPanel dataPanel = new JPanel();
37      dataPanel.setLayout(new BorderLayout(5, 10));
38
39      // Add x,y addresses of three defining
points of triangle
40      JPanel triangleTfGrid = new JPanel();
41      triangleTfGrid.setLayout(new GridLayout(4,
2, 5, 5));
42      triangleTfGrid.add(new JLabel("x"));
43      triangleTfGrid.add(new JLabel("y"));
44      // Add 2 x 3 array of JTextField
45      trianglePointsTf = new
ArrayList<PointXYTf>();
46      final int VERTEX_CNT = 3;
47
48      for (int i = 0; i < VERTEX_CNT; i++)
49      {
50          PointXYTf ptTf = new PointXYTf(new
Point2D.Double(0,0));
51          trianglePointsTf.add(ptTf);

```

```

52         triangleTfGrid.add(ptTf.getTfX());
53         triangleTfGrid.add(ptTf.getTfY());
54     }
55     dataPanel.add(triangleTfGrid,
BorderLayout.NORTH);
56
57     // Add 2 x 2 array of calculated results
58     JPanel results = new JPanel();
59     results.setLayout(new GridLayout(2, 2, 5,
5));
60     results.add(new JLabel("area"));
61     jtfArea = new JTextField(8);
62     jtfArea.setText("0.0");
63     area = 0;
64     results.add(jtfArea);
65     results.add(new JLabel("perimeter"));
66     jtfPerimeter = new JTextField(8);
67     jtfPerimeter.setText("0.0");
68     results.add(jtfPerimeter);
69     perimeter = 0;
70     dataPanel.add(results, BorderLayout.SOUTH);
71
72     add(dataPanel, BorderLayout.EAST);
73
74     // Add control panel to the bottom (SOUTH)
of the window
75     JPanel controls = new JPanel();
76     controls.setLayout(new
FlowLayout(FlowLayout.LEFT, 10, 20));
77     JButton jbtnCalc = new JButton("calculate");
78     jbtnCalc.addActionListener(
79         new ActionListener()
80     {

```

```

81         public void
actionPerformed(ActionEvent e)
82         {
83             calcTriangle();
84             repaint();
85         }
86     }
87 );
88     controls.add(jbtnCalc);
89     add(controls, BorderLayout.SOUTH);
90     triangle = new TriangleCartesian();
91     setVisible(true);
92 }
93
94 /**
95  * Calculate triangle's area and perimeter
96  */
97 private void calcTriangle()
98 {
99     Point2D.Double a, b, c;
100     System.out.println("Starting calcTriangle");
101     ArrayList<Point2D.Double> vertices = new
ArrayList<>();
102     for (PointXYTf row : trianglePointsTf)
103     {
104         System.out.println(row);
105         vertices.add(row.getPoint2D());
106     }
107     System.out.println("number of points: " +
vertices.size());
108     a = vertices.get(0);
109     b = vertices.get(1);
110     c = vertices.get(2);

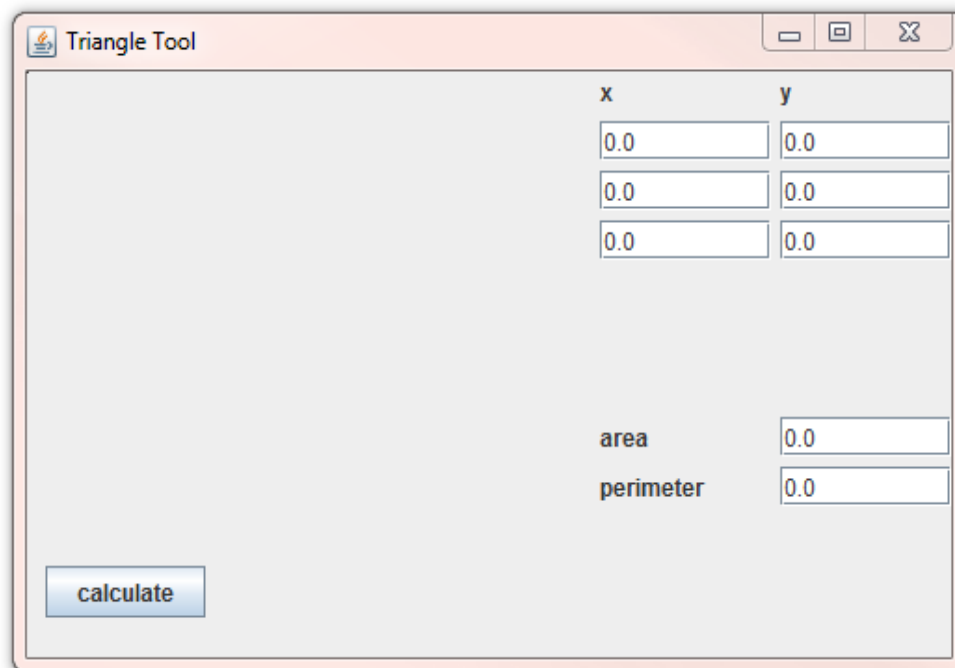
```

```
111         triangle =
112             new TriangleCartesian(a, b, c);
113         area = triangle.getArea();
114         String areaStr = String.format("%8.3f",
area);
115         jtfArea.setText(areaStr);
116         perimeter = triangle.getPerimeter();
117         String perimeterStr = String.format("%8.3f",
perimeter);
118         jtfPerimeter.setText(perimeterStr);
119     }
120
121
122     /**
123      * starts the program
124      */
125     private void go()
126     {
127
128     }
129
130     /**
131      * main method
132      */
133     public static void main(String[] args)
134     {
135         TriangleToolV4 tool = new TriangleToolV4();
136         tool.go();
137     }
138
139     /**
140      * inner class SketchPanel for drawing
triangles, etc
```

```

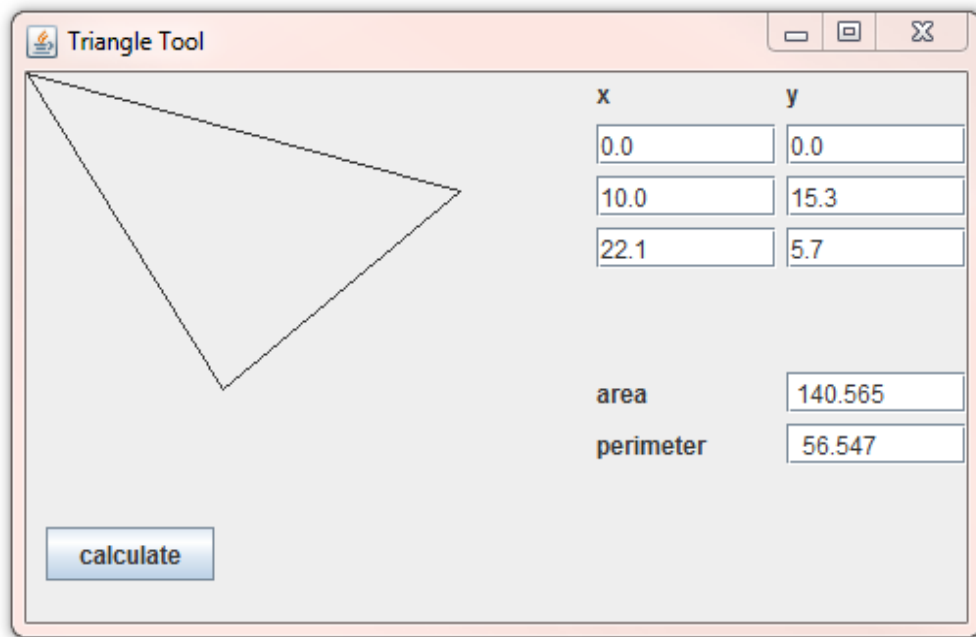
141      */
142      class SketchPanel extends JPanel
143      {
144
145          /**
146           * Override JPanel's paintComponent method
147           *
148           * @param Graphics object g
149           */
150          protected void paintComponent(Graphics g)
151          {
152              super.paintComponent(g);
153              triangle.drawTriangle(g);
154          }
155      }
156
157 } // end class TriangleTool

```



The final program produces this GUI.

After entering values for the three vertices the results are displayed.



The goal of this book is to demonstrate how java computer programs can help you discover new properties of mathematical objects. In this situation you would expand this simple program to display more properties of a triangle such as the inner circle touching all three sides and the outer circle touching all three vertices. Next you would add the ability to click and drag on a vertex so you can play with the triangle. Only then will constants be discovered.